Power BI Introduction: Tour of Power BI — Part 1

Power BI has been available for a number of years, but new functionality is added regularly. In this new series from Robert Sheldon, you will learn about the latest developments or get started with Power BI if you are new to it. Robert provides an overview of Power BI, the services and tools available in this article.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Microsoft's Power BI is not just a cloud service. It's a suite of integrated business intelligence (BI) tools for accessing and consolidating data and then presenting it as actionable insights. If you haven't checked out Power BI lately, you might not be aware of how aggressively Microsoft has been expanding its features and extending its reach into additional sources of data.

With Power BI, you can shape and transform data, aggregate and summarize data, apply complex calculations and conditional logic, and produce a wide range of visually rich reports that you can distribute to both internal and external users.

Power BI has, in fact, become a force to be reckoned with in the BI universe and is well worth trying out for yourself. Unfortunately, all the new and improved features, while in themselves have made Power BI a more robust offering, have also made it more difficult to understand how all the pieces fit together, a process helped little by the sometimes confusing and often obscure marketing hype.

In this article, the first in a series about Power BI, I try to make sense of the main components that are part of the Power BI ecosystem. The article takes a high-level look at these components in order to provide you with a foundation for delving into more specific details later in the series. I wrote several Simple Talk articles about Power BI back in 2015 and 2016, but much has changed this then, and this seems to be a good time to revisit the topic and fill in some of the newly added details.

Introducing Power BI

Microsoft describes Power BI as a "suite of business analytics tools that deliver insights throughout your organization." With Power BI, you can retrieve data from hundreds of data sources, shape the data to fit your specific requirements, perform ad hoc analytics, and present the results through various types of visualizations. Power BI greatly simplifies the entire BI process, making it possible for business users and data analysists to take control of their own reporting needs, while providing enterprise-grade security and scalability.

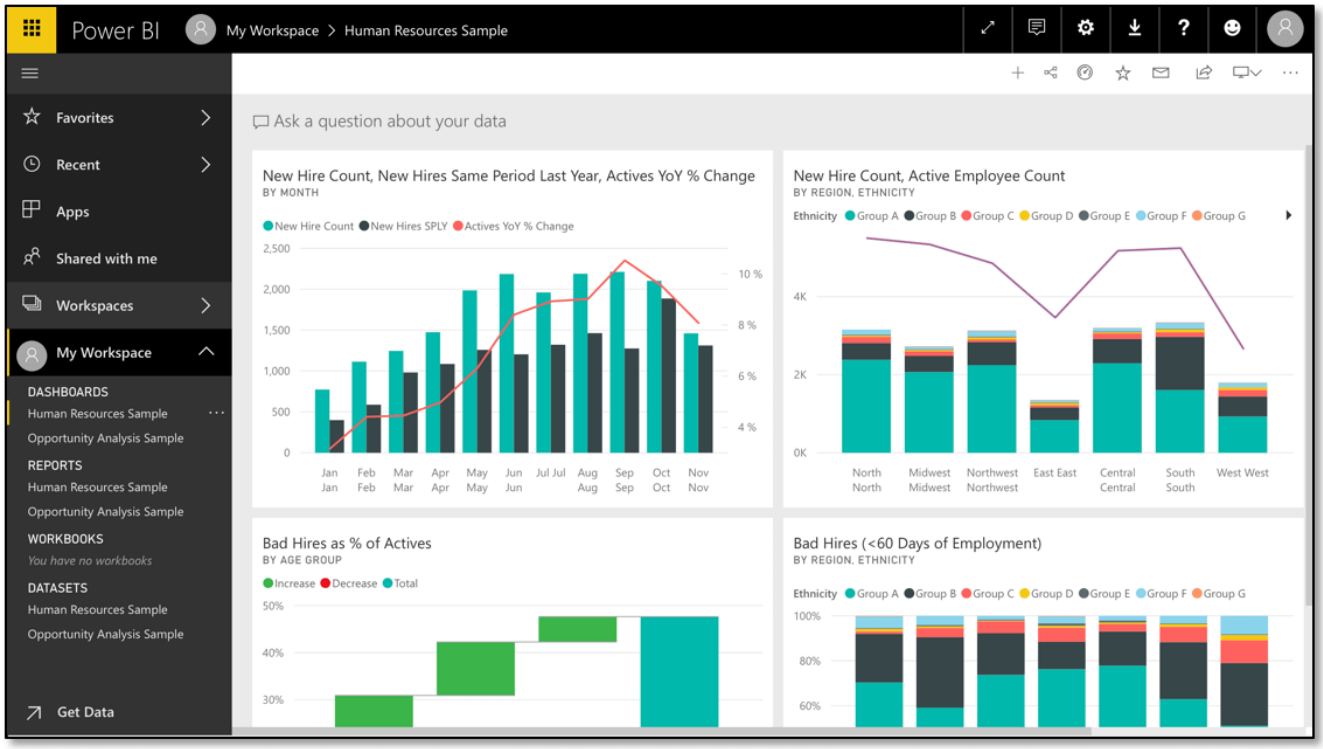
Microsoft makes Power BI available as part of the Microsoft Business Application Platform, a somewhat confusing umbrella term that refers to several related technologies, including Power BI, PowerApps, and (according to some documentation) Microsoft Flow. You should already have a sense of what Power BI is about, but you might not be familiar with the other two. PowerApps is a point-and-click application development platform, and Microsoft Flow is a workflow and business process management platform.

For this series, we're concerned primarily with Power BI, which provides a number of tools for delivering BI insights through browsers or mobile apps as well as embedding them within custom applications. In addition to the online service, Power BI includes Power BI Desktop, the Power BI mobile apps, the Power BI API, and Power BI Report Server. The rest of the article goes into more detail about each component.

Power BI Service

The Power BI service lies at the heart of the Power BI offering, providing a cloud-based platform for connecting to data and building reports. Users can access the service through a web-based portal that provides the tools necessary to retrieve, transform and present business data. For example, the following figure shows the portal with the *Human Resources Sample* dashboard

selected. The dashboard includes several visualizations that are part of the *Human Resources Sample* report. (Microsoft provides several sample datasets, reports, and dashboards for learning about Power BI.)



Notice that the *My Workspace* section in the left navigation pane is expanded, showing links to dashboards, reports, workbooks, and datasets. These four items represent the primary components that go into the Power BI presentation structure:

- **Dataset:** Collection of related data that you import or connect to. A dataset is similar to a database table and can be used in multiple reports, dashboards, and workspaces. You can retrieve data from files, databases, online services, or Power BI apps published by other people in your organization.
- **Report:** One or more pages of visualizations based on a single dataset. A report can be associated with only one workspace, but it can be associated with multiple dashboards within that workspace. You can interact with a report either in Reading view or Editing view, depending on your granted level of permissions.
- **Dashboard:** A presentation canvas that contains zero or more tiles or widgets. A dashboard can be associated with only one workspace, but it can display visualizations from multiple datasets or reports. You can pin an individual visualization to a tile or pin an entire report to a dashboard. If you're a Power BI Pro or Premium subscriber, you can also share dashboards.
- **Workspace:** A container for datasets, reports, and dashboards. The Power BI service supports two types of workspaces: *My Workspace* and app workspaces, which you access through the *Workspaces* section in the left navigation pane. *My Workspace* is a personal work area provided automatically when you log into the service. Only you can access this space. An app workspace is used to share and collaborate on content. You can also use an app workspace to create, publish, and manage Power BI apps (collections of dashboards and reports).

Microsoft offers several Power BI subscription plans. At the entry level is the Power BI Free service. To register, you must use a work email account, not a personal account such as Gmail. If you try, you'll receive a polite message denying you access. In addition, you're limited 10 GB of storage, and you can use only the basic features, although these are actually fairly robust. For example, you can connect to all the supported data sources, clean and prepare the data, and build and publish reports. You can even embed the reports in public websites.

The next level up is the Power BI Pro service, which builds on the Free service but adds such features as sharing, collaboration, auditing, and auto-refresh. The Pro service also lets users create app workspaces. As with the Free service, Pro users are limited to 10 GB of storage; however, they can also create app workspaces that support up to 10 GB of storage each. Microsoft currently offers of a 60-day free trial of the Pro service.

The Power BI Premium subscription level builds on the Pro service, but also provides an organization with dedicated resources (capacities) for deploying Power BI at scale, with up to 100 TB of storage per capacity. In addition, an organization can distribute Power BI content to non-licensed users as well as embed content in customized applications. Plus, the Premium service includes Power BI Report Server, an on-premises solution for publishing reports in-house.

Microsoft also offers versions of the Power BI service for US government customers and European Union customers. The services are separate from the regular commercial services. Microsoft does not offer a free version of either one. (Contact Microsoft for more details.)

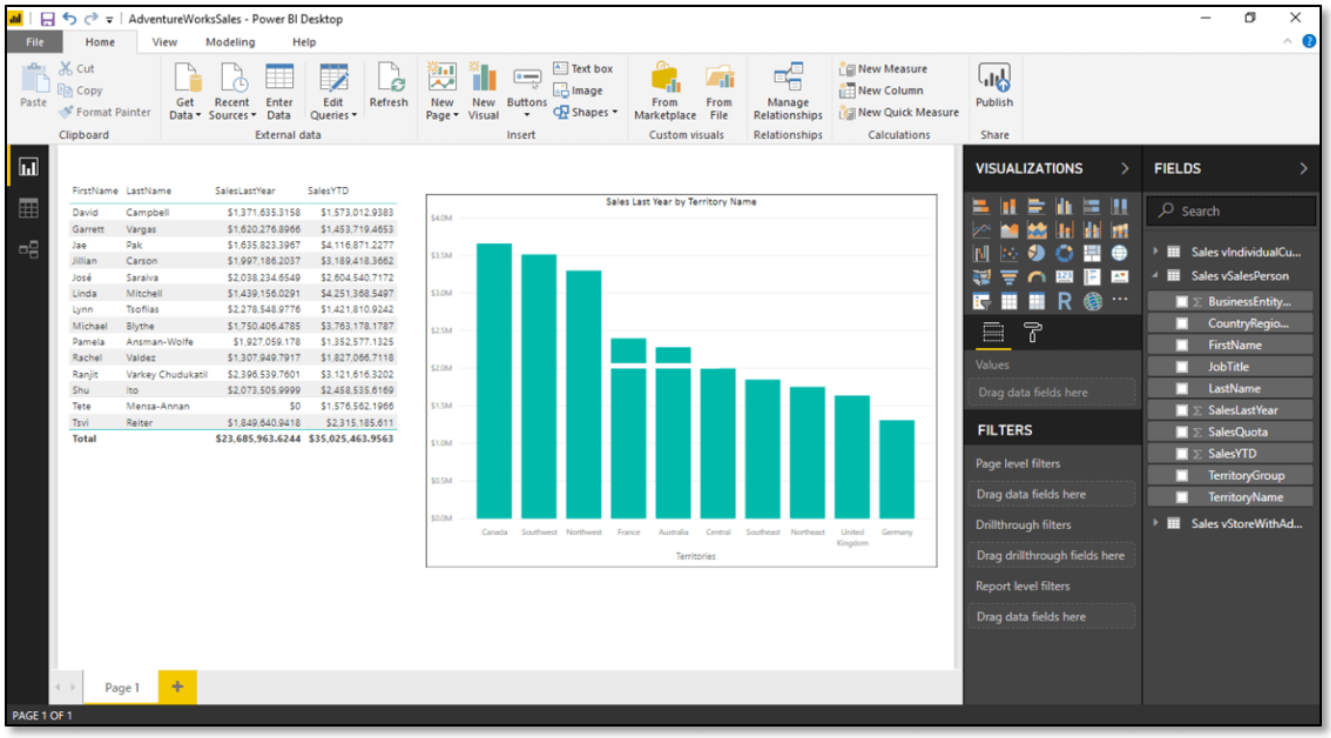
Power BI Desktop

Power BI Desktop is a downloadable application that Microsoft provides for free. The application is essentially a report-building tool that provides capabilities similar to the Power BI service, but kicks them up a notch. With Power BI Desktop, you can build advanced data queries and models, create sophisticated reports and visualizations, and publish the consolidated report packages to the Power BI service or Power BI Report Server.

Both conceptually and physically, Power BI Desktop can be divided into three categories, or views, for how you interact with data and create reports:

- **Report view:** A canvas for building and viewing reports based on the datasets defined in *Data* view.
- **Data view:** Defined datasets based on data retrieved from one or more data sources. *Data* view offers limited transformation features, with many more capabilities available through the *Query Editor*, which opens in a separate window.
- **Relationships view:** Identified relationships between the datasets defined in *Data* view. When possible, Power BI Desktop identifies the relationships automatically, but you can also define them manually.

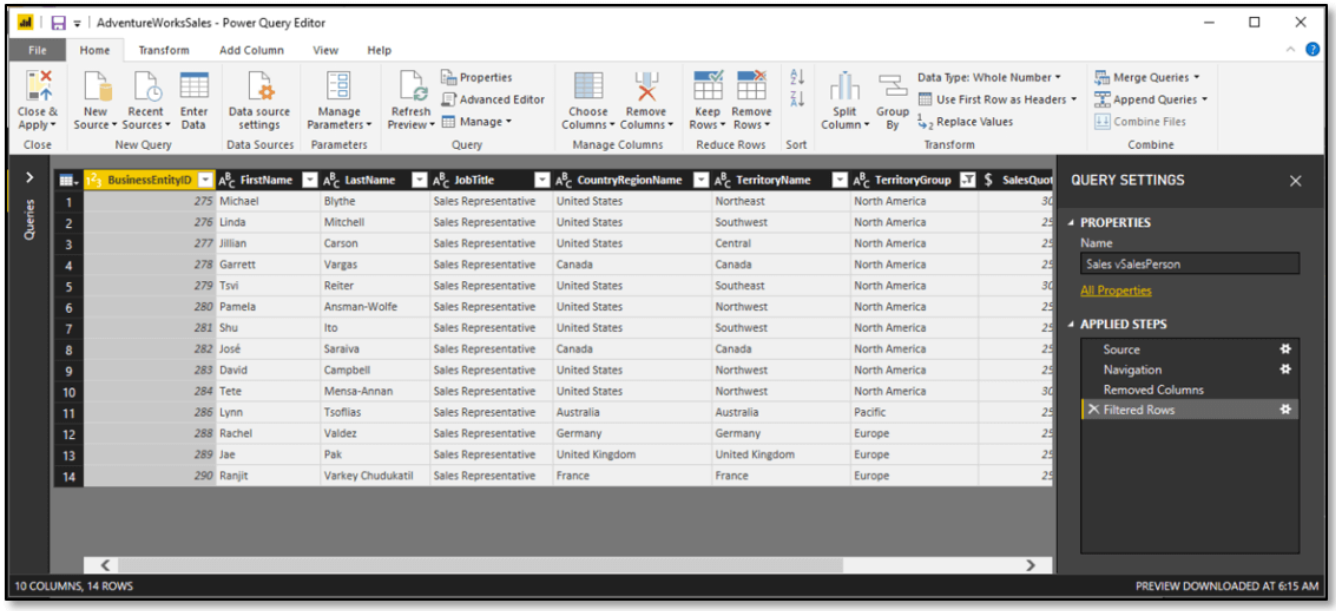
To access any of the three views, click the applicable button in the navigation pane at the left side of the Power BI Desktop interface, shown in the following figure. In this case, *Report* view is selected, displaying a one-page report that includes two visualizations, one table and one bar chart.



The data for the report comes from the *AdventureWorks2017* sample database, running on a local instance of SQL Server 2017. However, you can define datasets based on data from a variety sources, including files such as Excel, CSV, XML, and JSON; databases such as Oracle, Access, DB2, and MySQL; and online services such as Azure, Salesforce Reports, Google Analytics, and Facebook.

Power BI Desktop also provides generic connectors for accessing data not available through the predefined connectors. For example, you can use an interface type such as ODBC, OLE DB, OData, or REST to connect to a data source, or you can run an R script and create a dataset based on the results.

Where Power BI Desktop really shines, when compared to the Power BI service, are in the features available in the *Query Editor* to shape and combine data, some of which are shown in the following figure. In this case, the *Sales.vSalesPerson* dataset is open, which is based on a view in the *AdventureWorks2017* database with the same name.



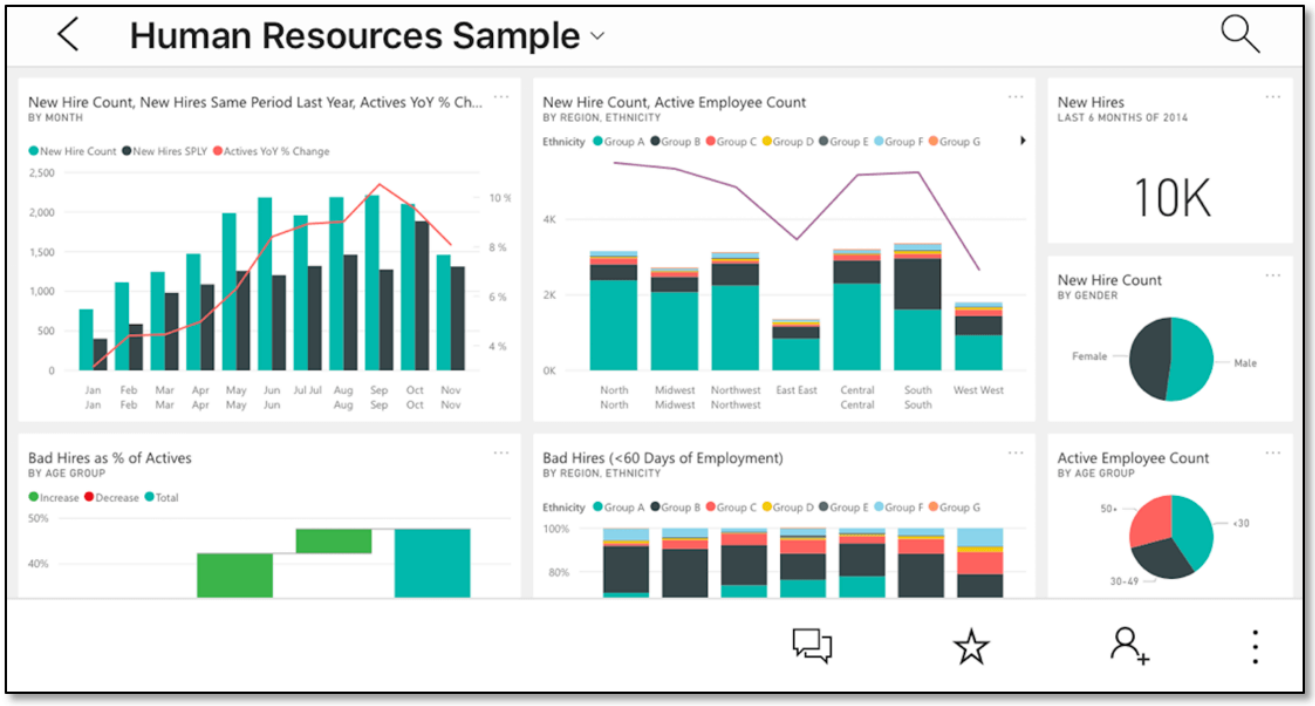
In the *Query Editor*, you can rename datasets or columns, filter out columns or rows, aggregate or pivot data, and shape data in numerous other ways. You can also combine datasets, even if they come from different sources. In addition, Power BI Desktop provides the Data Analysis Expressions (DAX) language for performing more complex transformations.

After you've gotten the data in the format you need, you can use *Report* view to create multiple types of visualizations, including bar chats, line charts, scatter charts, pie charts, treemaps, tables, matrices, and maps. *Report* view provides numerous options for configuring and refining the charts so you're presenting the data as effectively as possible. In addition, you can import and display key performance indicators (KPIs) as well as add dynamic reference lines to visualizations to focus on important insights. Once you have your reports the way you want them, you can publish them to the Power BI service or to Power BI Report Server.

There are plenty of other features available to Power BI Desktop than what I've covered here, and most of them are easy to access and understand. The UI is powerful and intuitive enough to support a wide range of users, from data stewards to business users to data analysts.

Power BI Mobile Apps

Microsoft offers Power BI mobile apps for iOS, Android, and Windows mobile devices. The apps make it possible to provide specific users with access the Power BI dashboards, reports, and apps, while taking into account the form factor of the smaller devices. For example, the following figure shows the *Human Resources Sample* dashboard (in landscape mode), as it is rendered by the Power BI app for iPhone.



With a Power BI app, you can connect to either the Power BI service or to a Power BI Report Server instance. Because you're dealing with an app rather than a website, you can view the Power BI content offline. Once you're reconnected, Power BI automatically refreshes the data. When you're connected via a 3G network, the data is refreshed every 24 hours. When you're connected via Wi-Fi, the updates occur every two hours.

A Power BI app lets you zoom in on individual visualizations, add annotations, and share snapshots of a report or visualization. You can also filter content by owner, search content, or tag content as favorites. With a Power BI Pro or Premium license, you can share a link with colleagues so they can view your dashboards. There are plenty of other features as well, and since the apps are free, there's no reason not to try one out, as long as you're signed up for the Power BI service.

When you're creating Power BI reports, you can optimize them for mobile devices. This causes Power BI to add features to the reports specific to mobile usage, such as allowing users to drill down into visualizations. In addition, you can add slicers to your reports that let users filter the displayed data. Plus, you can create a QR code for a report and distribute it to colleagues, who can then scan the code from within their Power BI app to view the report.

Although the mobile apps are similar from one platform to the next, there are some differences. For example, only the iOS and Android apps let users annotate visualizations, share snapshots, or view a report via a QR code. Despite these differences, the main functionality is the basically the same from one platform to the next, allowing users to view a wide range of information, no matter where they're working or how they're connected.

Organizations that use Microsoft Intune to manage mobile devices can also use the service for managing the Power BI mobile apps. By configuring the necessary policies, administrators can control how data is handled and when application data should be encrypted.

Power BI API

Microsoft offers development teams a REST API that provides programmatic access to Power BI resources. Developers can use the API with any programming language that supports REST calls.

One of the most important capabilities that the API supports is the ability to embed reports, tiles, and dashboards into customized applications. The reports are fully interactive and are automatically refreshed whenever the data changes. Depending on the organization's subscription level, developers can embed components into applications for internal users who are licensed for Power BI or for users who do not have Power BI accounts.

Another option for developers using the API is the ability to create customized visualizations that can be used in Power BI reports. Customized visualizations are written in TypeScript, a JavaScript superset that Microsoft developed for large-scale JavaScript

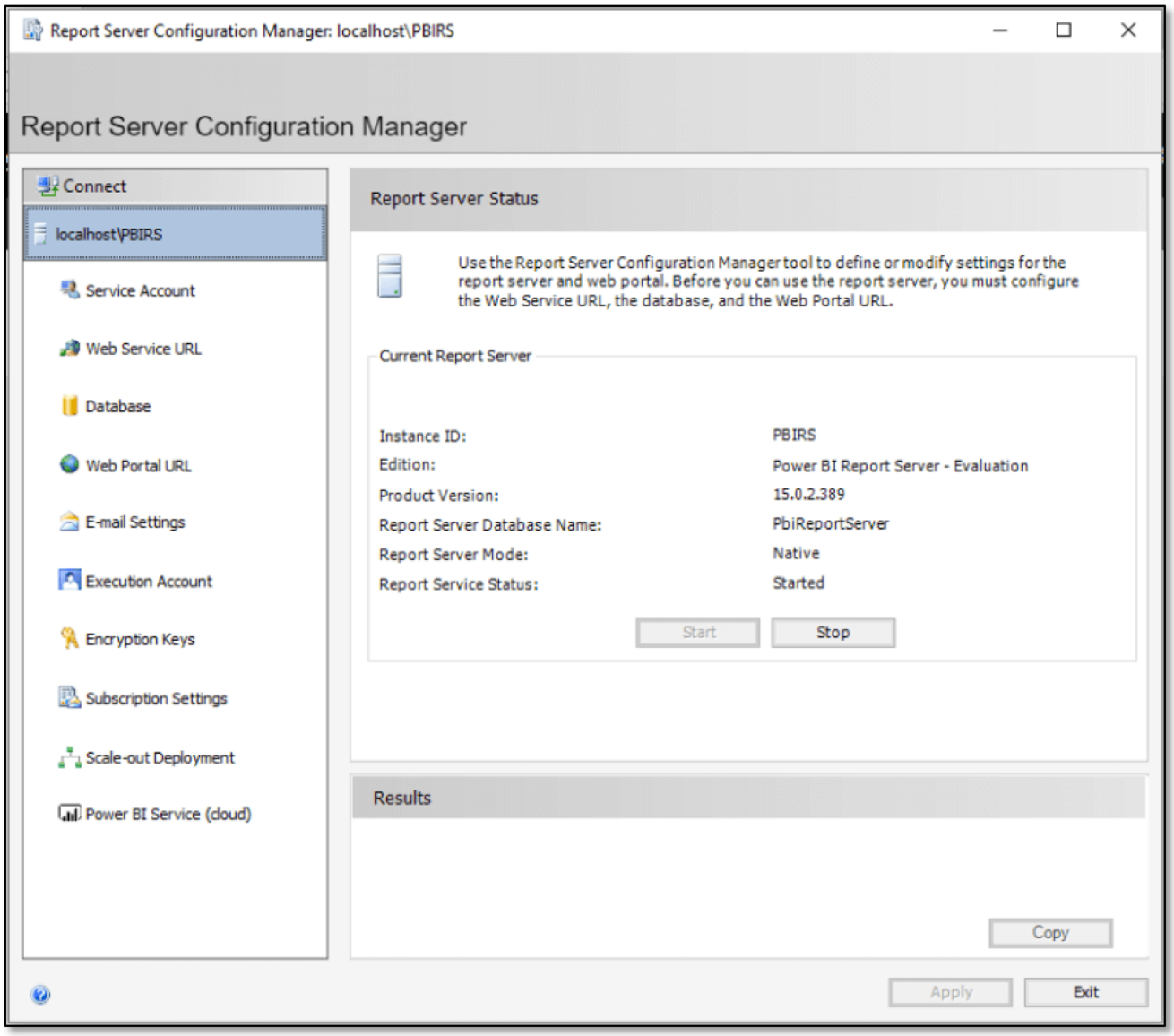
applications. The visualizations also incorporate cascading style sheets (CSS) and support such features as variables, nesting, mixins, loops, and conditional logic.

Developers can also use the Power BI API to push data into a dataset. In this way, they can extend their business workflows to the Power BI environment. Any reports or dashboards that incorporate the dataset are automatically updated to reflect the new data.

Power BI Report Server

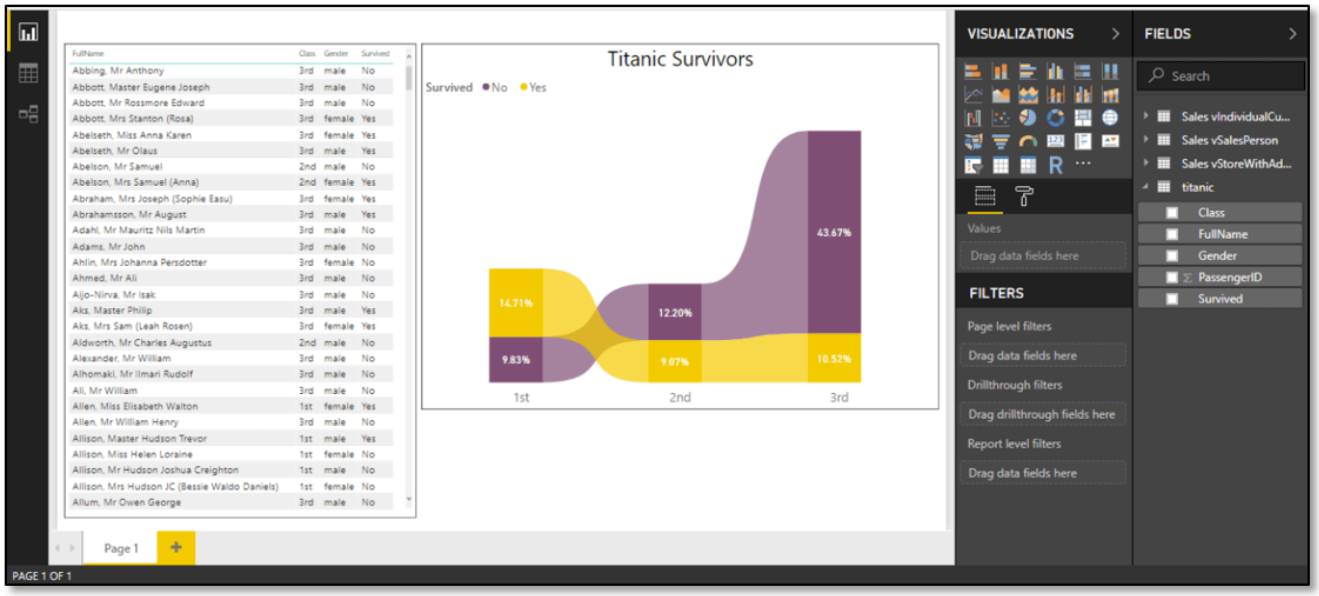
One of the newer tools in the Power BI arsenal is Power BI Report Server, an on-premises solution for creating, deploying and managing Power BI reports. The product is included with a Power BI Premium subscription to provide customers with a tool for delivering reports from within their own data centers. Users, in turn, can view the reports via their browsers or Power BI mobile apps or as email attachments.

If you choose to install Power BI Report Server, you must use the Report Server Configuration Manager to specify such settings as the service account, web service URL, SQL Server database, and web portal URL, as shown in the following figure. You need to set up your configurations before you can start working with the actual reports.



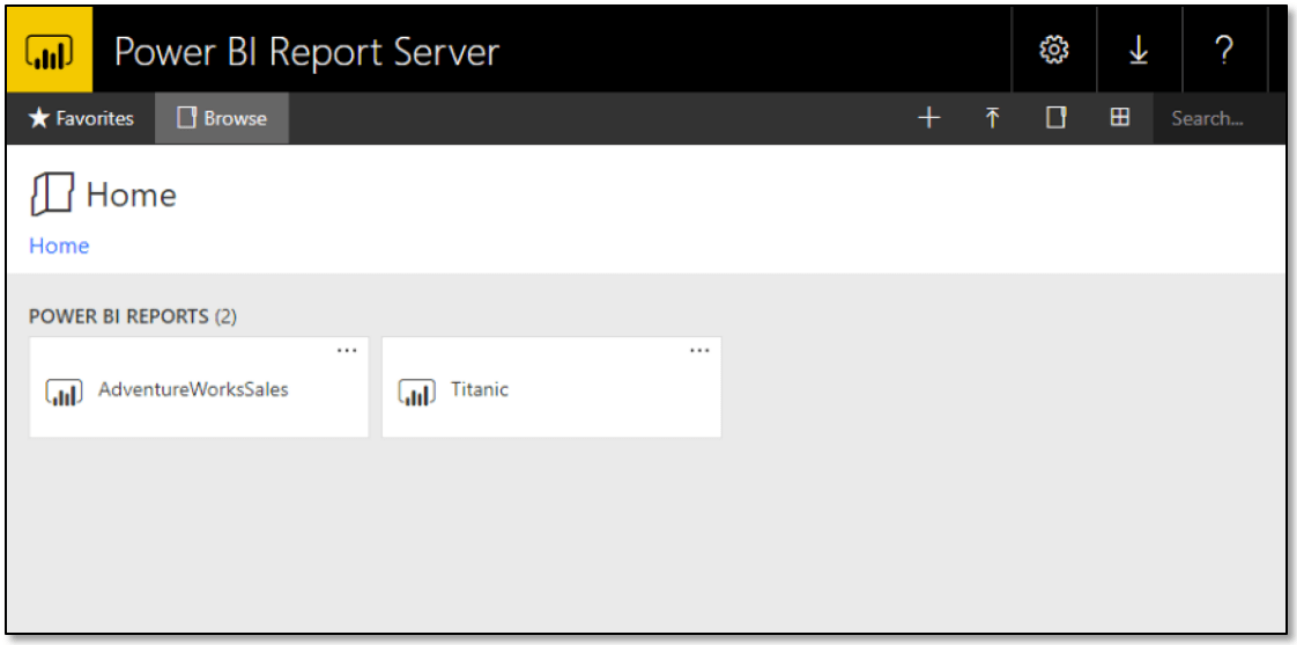
The Report Server Configuration Manager is included in the Power BI Report Server installation, but it is separate from the tools you use to manage the reports. For report management, you must use the Report Server web portal, which is enabled after you configure the necessary settings. Through the web portal, you can access all your reports and KPIs, as well as carry out such tasks as schedule data updates or subscribe to published reports.

Like the Power BI service, Power BI Report Server works in conjunction with Power BI Desktop. You can create reports and then save them to Power BI Report Server. For example, you can save a report such as the one shown in the following figure to Power BI Report Server.

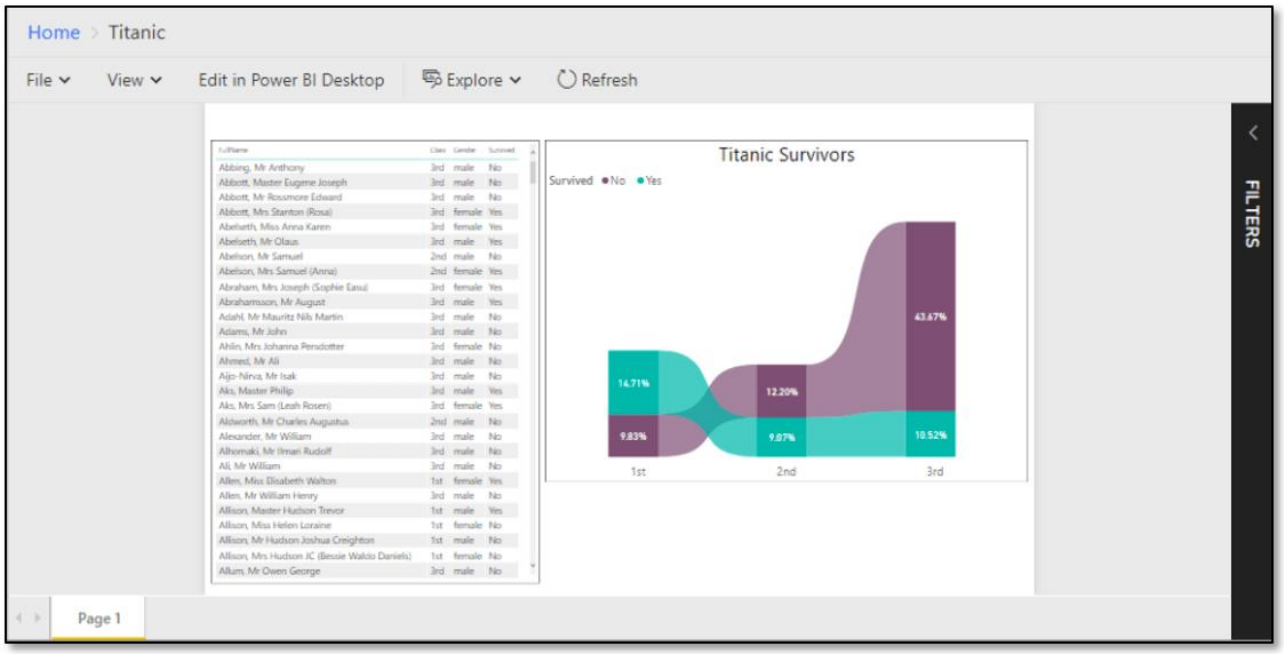


The report is based on data from the Titanic dataset, available as a CSV file from the site https://vincentarelbundock.github.io/Rdatasets/datasets.html. The report includes one table and one ribbon visualization. To save the report to Power BI Report Server, you must use the *Save As* command and provide the web portal URL.

When you connect to the Power BI Report Server web portal, you're taken to your *Home* page, which lists any reports you've added to the server. For example, the following figure shows the two reports I added on my system: *AdventureWorksSales* and *Titanic*. I had created both of these reports in Power BI Desktop.



To view the *Titanic* report, click the applicable report icon. This takes you to the report page shown in the following figure.



Notice that the report looks similar to what we saw in Power BI Desktop, although the colors are a bit different from the original. Even so, this should give you an idea of how Power BI Report Server works and how easy it is to copy your reports from Power BI Desktop to Power BI Report Server.

Keep in mind, however, that Power BI Report Server is still a young product and, as such, you might run into some odd behavior along with way. For example, if you install the product on a standalone server that is not part of a domain, you might have trouble viewing reports in the Edge browser. Worse still, the error messages you receive might send you down a rabbit hole that can waste much of your time (as it did me). However, the Chrome browser appears to work fine for viewing reports in Power BI Report Server, and I've also read that running Internet Explorer as an administrator can get the reports to render properly.

Another problem you might run into has to do with the Power BI Desktop version that you're using. You must use one that's optimized for Power BI Report Server, which is not always the most current release. This can be problematic if you create reports in a newer version of Power BI Desktop and then discover you have to revert back to an older version to save the reports to Report Server. The older version of Power BI Desktop might not be able to properly process the report files. Last I checked, the most recent Power BI Desktop release was April 2018, but Power BI Report Server required a March 2018 release of Power BI Desktop.

I've no doubt that, with time, Microsoft will get many of these bugs worked out and will be adding new features along the way. It will be interesting to see what Power BI Report Server looks like a year from now, or even over the next six months.

More to Come

As Power BI continues to evolve and grow, new features will continue to come online, as well as services related to the Power BI ecosystem. For example, Microsoft now offers Power BI Embedded, a service for ISVs and developers looking for an easier way to embed Power BI analytics into their applications. Microsoft also now provides features for better integrating Excel and Power BI. In addition, Microsoft will soon be offering *insights,* line-of-business apps that apply advanced intelligence to data for better understanding that data. Two insight apps—Power BI for Sales Insights and Power BI for Service Insights—are expected to preview in the very near future. What will come after that is anyone's guess.

Power BI Introduction: Working with Power BI Desktop — Part 2

Power BI Desktop is the free tool from Microsoft for developing in Power BI. In this article, Robert Sheldon walks you through importing and manipulating data with the Query Editor in Power BI Desktop.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1

2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Microsoft's Power BI is a comprehensive suite of integrated business intelligence (BI) tools for retrieving, consolidating, and presenting data through a variety of rich visualizations. One of the suite's most important tools is Power BI Desktop, a free, on-premises application for building comprehensive reports that can be published to the Power BI service or saved to Power BI Report Server, where they can be shared with users through their browsers, mobile apps, or custom applications.

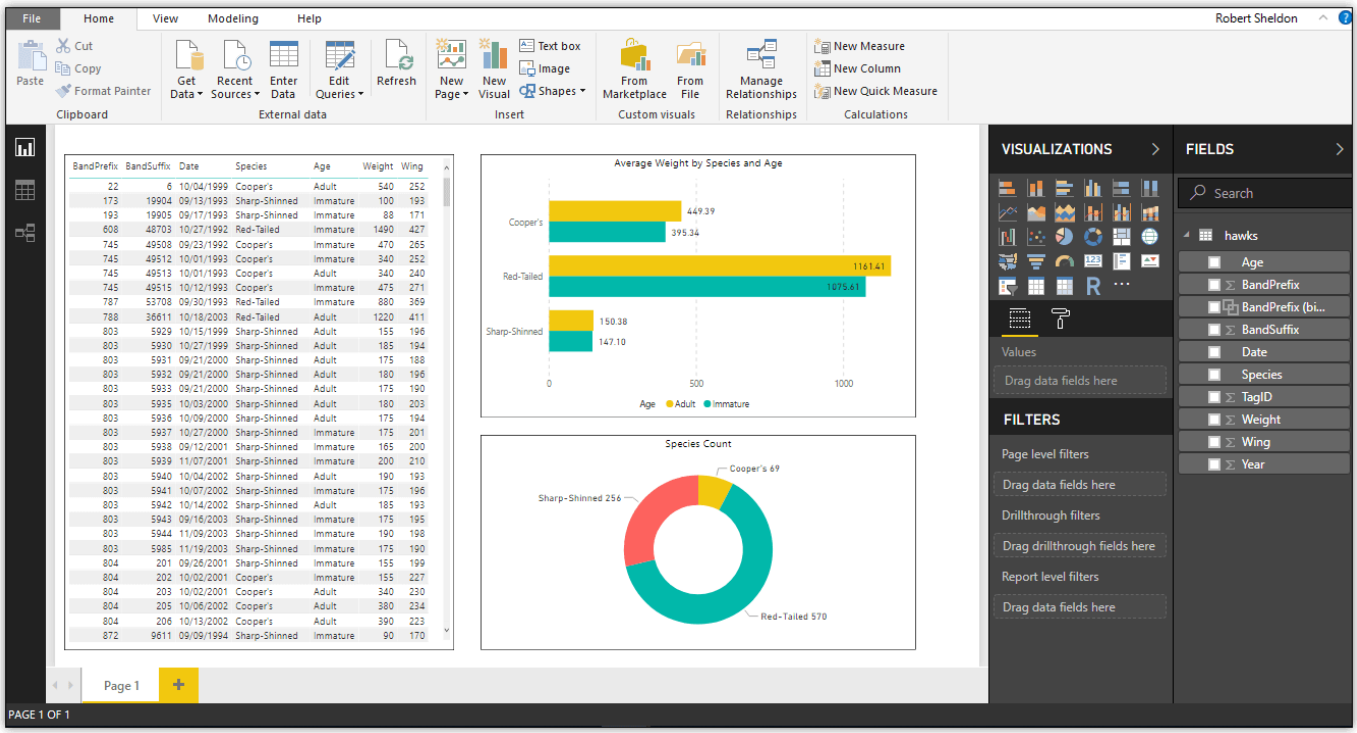
This article is the second in a series about Power BI. The first article covered the basic components that make up the Power BI suite. This article focuses specifically on Power BI Desktop because of the important role it plays in creating reports that can then be used by the other components.

With Power BI Desktop, you can access a wide range of data sources, consolidate data from multiple sources, transform and enhance the data, and build reports that utilize the data. Although you can access data sources and create reports directly in the Power BI service, Power BI Desktop provides a far more robust environment for working with data and creating visualizations. You can then easily publish your reports to the Power BI service by providing the necessary credentials.

The features in Power BI Desktop are organized into three views that you access from the navigation pane at the left side of the main window:

- **Report view:** A canvas for building and viewing reports based on the datasets defined in *Data* view.
- **Data view:** Defined datasets based on data retrieved from one or more data sources. *Data* view offers limited transformation features, with many more capabilities available through Query Editor, which opens in a separate window.
- **Relationships view:** Identified relationships between the datasets defined in *Data* view. When possible, Power BI Desktop identifies the relationships automatically, but you can also define them manually.

To access any of the three views, click the applicable button in the left navigation pane. The following figure shows Power BI Desktop with *Report* view selected, displaying a one-page report with three visualizations.



When working in Power BI Desktop, you will normally perform the following four basic steps, although not necessarily in a single session:

1. Connect to one or more data sources and retrieve the necessary data.
2. Transform and enhance the data using *Data* view, *Relationships* view, or the Query Editor, as necessary.
3. Create reports based on the transformed data, using *Report* view.
4. Publish the reports to the Power BI service or upload them to Power BI Report Server.

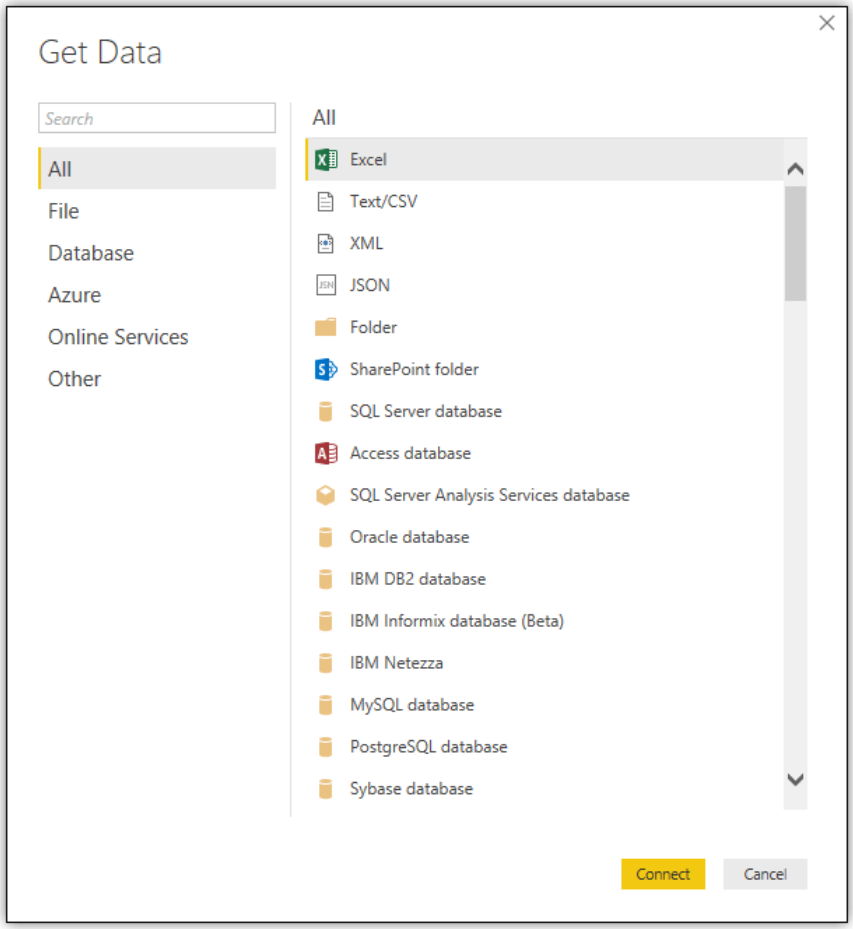
This article will focus primarily on steps 1 and 2 so you can get a better sense of how to retrieve and prepare the data, which is essential to building comprehensive reports. Later in the series, you'll learn more about building reports with different types of visualizations, but first you need to make sure you get the data right.

Connecting to Data in Power BI Desktop

Power BI Desktop supports connectivity to a wide range of data sources, which are divided into the following categories:

- **All:** Every data source type available through Power BI.
- **File:** Source files such as Excel, CSV, XML, or JSON.
- **Database:** Database systems such as SQL Server, Oracle Database, IBM DB2, MySQL, SAP HANA, and Amazon Redshift.
- **Azure:** Azure services such as SQL Database, SQL Data Warehouse, Blob Storage, and Data Lake Store.
- **Online Services:** Non-Azure services such as Google Analytics, Salesforce Reports, Facebook, Microsoft Exchange Online, and the Power BI service.
- **Other:** Miscellaneous data source types such as Microsoft Exchange, Active Directory, Hadoop File System, ODBC, OLE DB, and OData Feed.

To retrieve data from one of these data sources, click the *Get Data* button on the *Home* ribbon in the main Power BI window. This launches the *Get Data* dialog box, shown in the following figure.

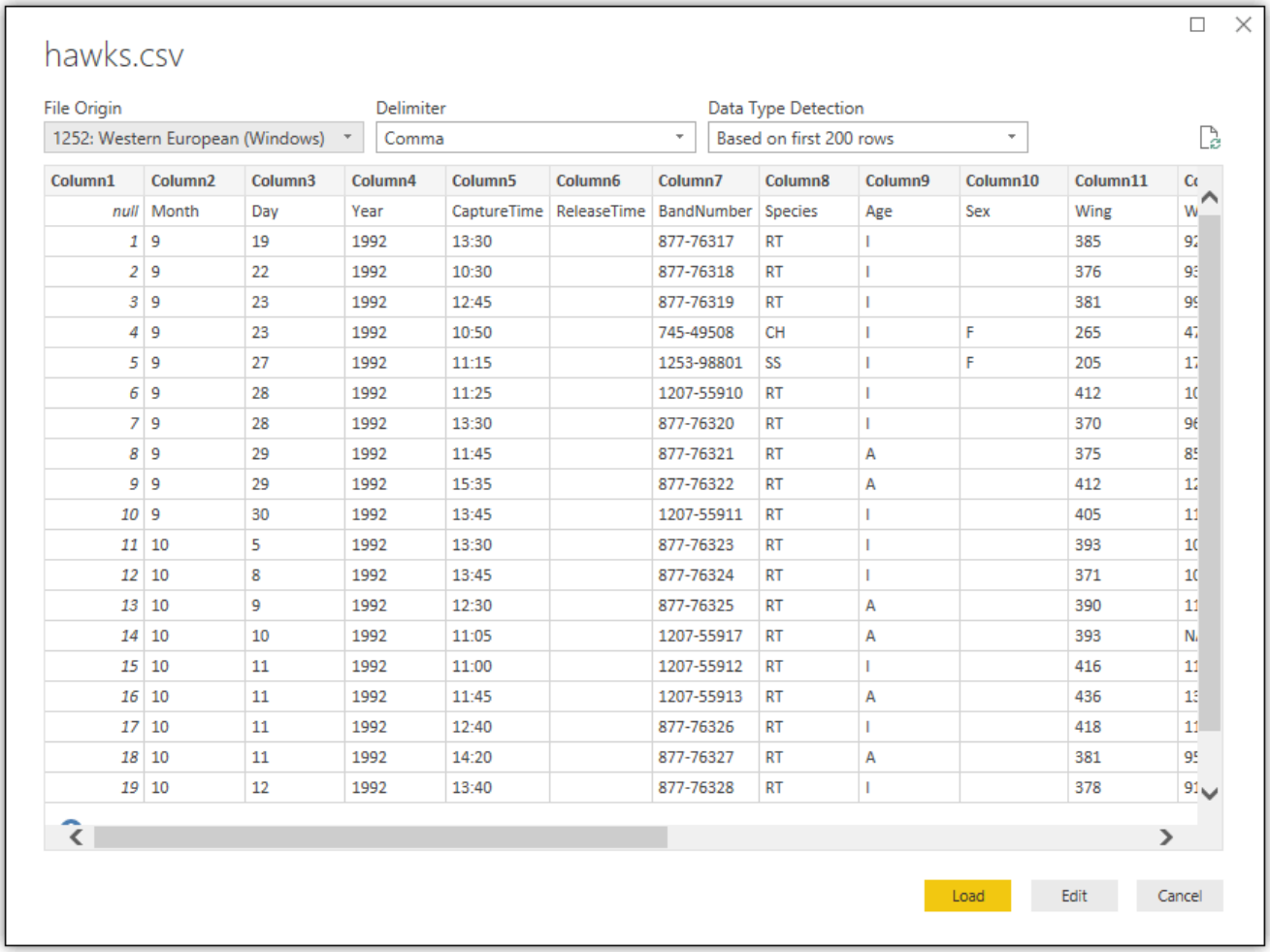


To access a data source, navigate to the applicable category, select the source type, and click *Connect*. You will then be prompted to provide additional information, depending on the data source. This might include connectivity details, an instance or

file name, or other types of information. After you provide the necessary details, Power BI Desktop will launch a preview window that will display a sample of the data along with additional options.

This article uses a CSV file based on data from the Hawks dataset, which is available through a GitHub collection of sample datasets. The Hawks dataset contains data collected over several years from a hawk blind at Lake MacBride near Iowa City, Iowa. Name the file *hawks.csv* and save it to a local folder (*C:\DataFiles*).

To import the data from the file into Power BI Desktop, select the *Text/CSV* data source type in the *Get Data* dialog box. Click *Connect* and the *Open* dialog box will appear. From there, navigate to the *C:\DataFiles* folder, select the *hawks.csv* file, and click *Open.* This launches the preview window shown in the following figure.



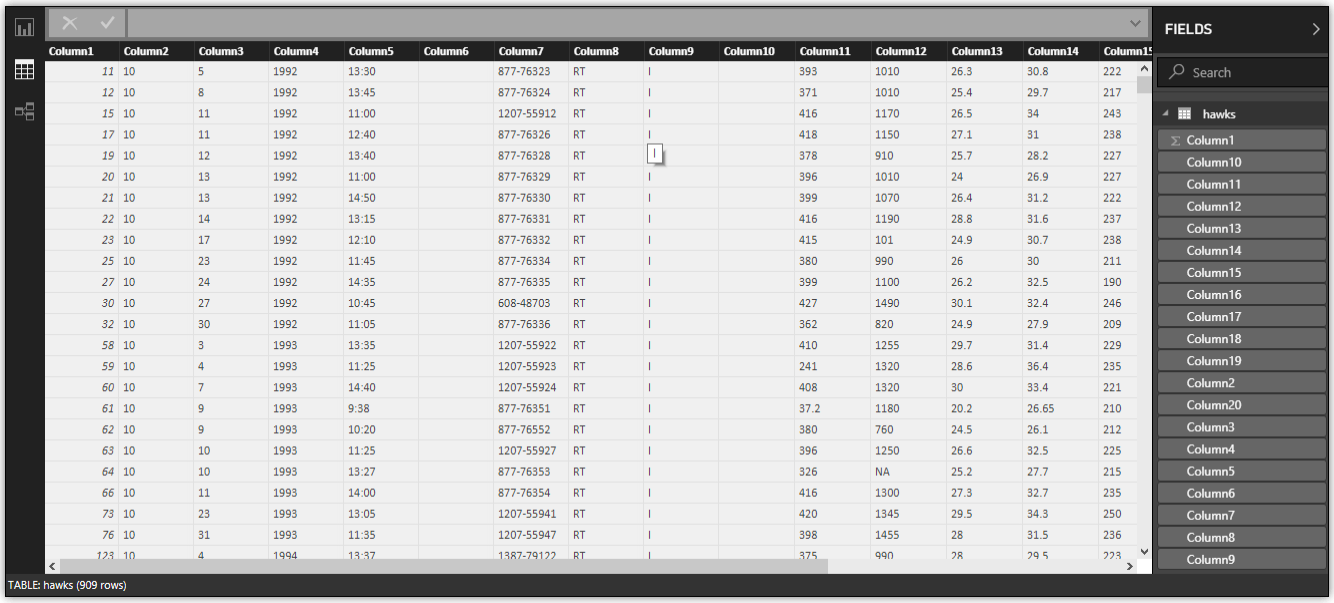
Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	Column11	Co
null	Month	Day	Year	CaptureTime	ReleaseTime	BandNumber	Species	Age	Sex	Wing	W
1	9	19	1992	13:30		877-76317	RT	I		385	92
2	9	22	1992	10:30		877-76318	RT	I		376	93
3	9	23	1992	12:45		877-76319	RT	I		381	99
4	9	23	1992	10:50		745-49508	CH	I	F	265	47
5	9	27	1992	11:15		1253-98801	SS	I	F	205	17
6	9	28	1992	11:25		1207-55910	RT	I		412	10
7	9	28	1992	13:30		877-76320	RT	I		370	96
8	9	29	1992	11:45		877-76321	RT	A		375	85
9	9	29	1992	15:35		877-76322	RT	A		412	12
10	9	30	1992	13:45		1207-55911	RT	I		405	11
11	10	5	1992	13:30		877-76323	RT	I		393	10
12	10	8	1992	13:45		877-76324	RT	I		371	10
13	10	9	1992	12:30		877-76325	RT	A		390	11
14	10	10	1992	11:05		1207-55917	RT	A		393	N
15	10	11	1992	11:00		1207-55912	RT	I		416	11
16	10	11	1992	11:45		1207-55913	RT	A		436	13
17	10	11	1992	12:40		877-76326	RT	I		418	11
18	10	11	1992	14:20		877-76327	RT	A		381	95
19	10	12	1992	13:40		877-76328	RT	I		378	91

In addition to being able to preview a subset of the data, you can configure several options: *File Origin* (the document's encoding), *Delimiter,* and *Data Type Detection*. In most cases, you'll be concerned primarily with the *Data Type Detection* option. By default, this is set to *Based on first 200 rows,* which means that Power BI Desktop will convert the data to Power BI data types based only on the first 200 rows of data.

This might be okay in some cases, but there could be times when a column includes a value different from the majority of values, but that value is not in the first 200 rows. As a result, you could end up with a suboptimal data type or even an error. To avoid this risk, you can use the *Data Type Detection* option to instruct Power BI Desktop to base the data type selection on the entire dataset, or you can instead forego any conversion and keep all the data as string values.

You can also edit the dataset before loading it into Power BI Desktop. When you click the *Edit* button, Power BI Desktop launches Query Editor, which provides a number of tools for transforming data. After you make the changes, you can then save the updated dataset.

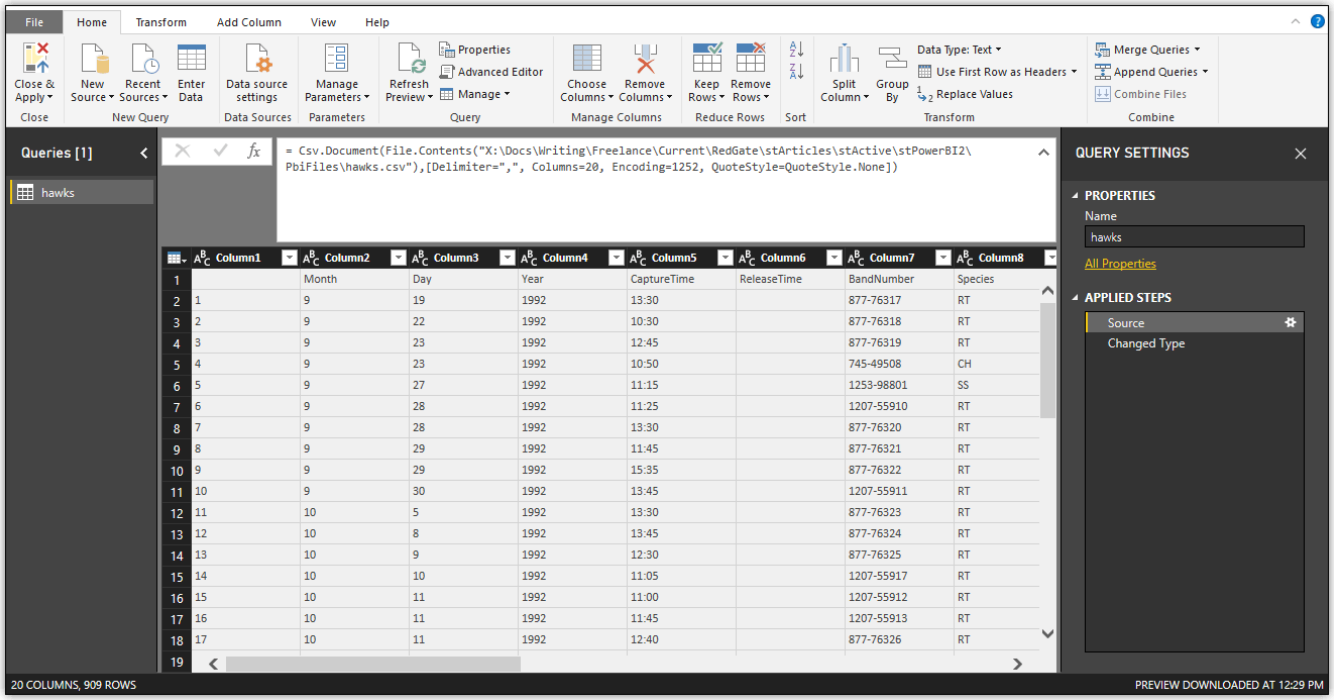
In this case, load the data into Power BI Desktop without changing any settings or editing the dataset. You will be transforming the data later as part of working through this article. To import the data as is, you need only click the *Load* button. You can then view the imported dataset in *Data* view, as shown in the following figure.



After you import a dataset into Power BI Desktop, you can use the data in your reports as is, or you can edit the data, applying various transformations and in other ways shaping the data. Although you can make some modifications in *Data* view, such as being able to add or remove columns or sort data, for the majority of transformations, you will need to use Query Editor, which includes a wide range of features for shaping data.

Introducing Query Editor

To launch Query Editor, click the *Edit Queries* button on the *Home* ribbon. Query Editor opens as a separate window from the main Power BI Desktop window and displays the contents of the dataset selected in the left pane, as shown in the following figure. In this case, the pane includes only the *hawks* dataset, which is the one you just imported.



The dataset itself is displayed in the main pane of the Query Editor window. You can scroll up and down or right and left to see all the data. The pane to the right, which is separated into two sections, provides additional information about the dataset.
The *Properties* section displays the dataset's properties. In this case, the section includes only the *Name* property. To view all properties associated with a dataset, click the *All Properties* link within that section.

The *Applied Steps* section lists the transformations that have been applied to the dataset in the order they were applied. Each step in this section is associated with a script statement written in the Power Query M formula language. The statement is shown in the formula bar directly above the dataset. (If the formula bar is not displayed, select the *Formula Bar* checkbox on the *View* ribbon to display it.)

Currently, the *Applied Steps* section for the *hawks* dataset includes only two steps: *Source* and *Changed Type*. Both steps are generated by default when importing a CSV file. The *Source* step corresponds to the following M statement (provided here in case you cannot read it in the above figure):

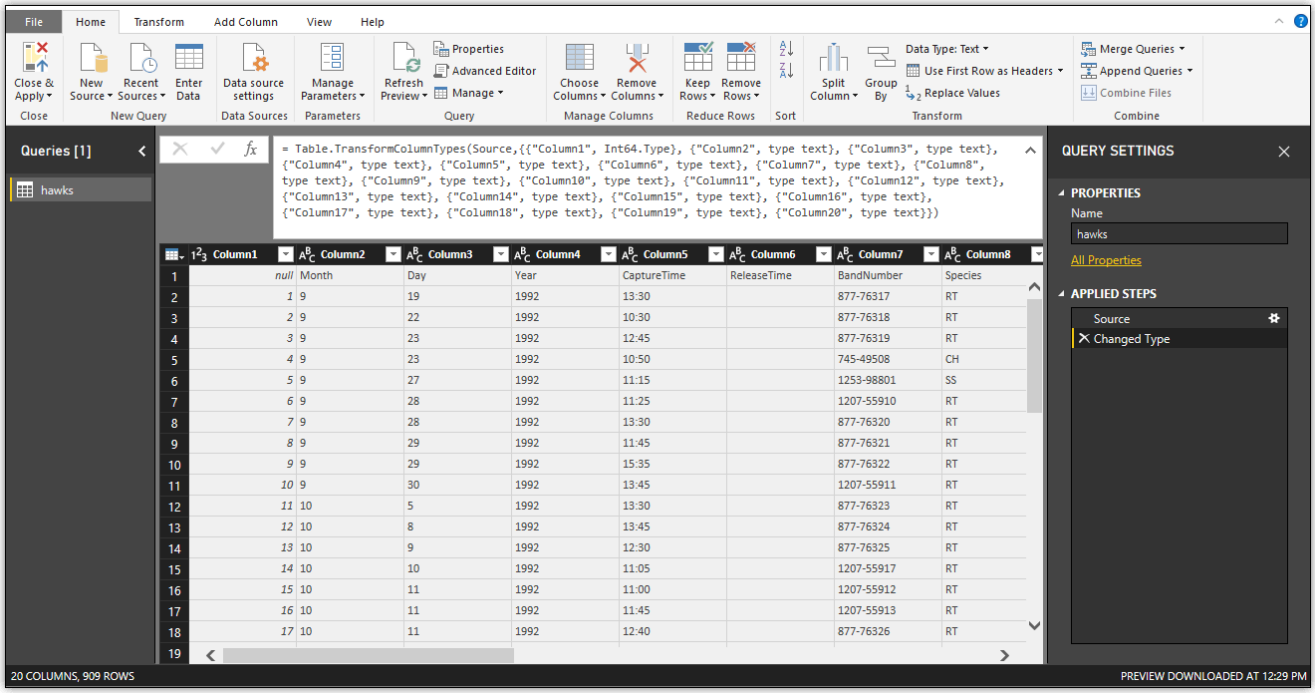
```
= Csv.Document(File.Contents("C:\DataFiles\hawks.csv"),[Delimiter=",", Columns=20, Encoding=1252, QuoteStyle=QuoteStyle.None])
```

The M statement consists primarily of the connection string necessary to import the *hawks.csv* file. Notice that it includes the delimiter and encoding specified in the preview window, along with other details.

The second step, *Changed Type,* converts the data in each column to Power BI data types. The step is associated with the following M statement, which shows that the first column is assigned the **Int64** type and that all other columns are assigned the **text** type:

```
= Table.TransformColumnTypes(Source,{{"Column1", Int64.Type}, {"Column2", type text}, {"Column3", type text}, {"Column4", type text}, {"Column5", type text}, {"Column6", type text}, {"Column7", type text}, {"Column8", type text}, {"Column9", type text}, {"Column10", type text}, {"Column11", type text}, {"Column12", type text}, {"Column13", type text}, {"Column14", type text}, {"Column15", type text}, {"Column16", type text}, {"Column17", type text}, {"Column18", type text}, {"Column19", type text}, {"Column20", type text}})
```

If you had opted to forego data type conversions when importing the *hawks.csv* file, all columns would be assigned the **text** data type, but you instead stuck with the *Based on first 200 rows* option, which resulted in one column being converted to the **Int64** type. The following figure shows the dataset with the *Changed Type* step selected and its associated M statement in the formula bar.



	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
1	null	Month	Day	Year	CaptureTime	ReleaseTime	BandNumber	Species
2	1	9	19	1992	13:30		877-76317	RT
3	2	9	22	1992	10:30		877-76318	RT
4	3	9	23	1992	12:45		877-76319	RT
5	4	9	23	1992	10:50		745-49508	CH
6	5	9	27	1992	11:15		1253-98801	SS
7	6	9	28	1992	11:25		1207-55910	RT
8	7	9	28	1992	13:30		877-76320	RT
9	8	9	29	1992	11:45		877-76321	RT
10	9	9	29	1992	15:35		877-76322	RT
11	10	9	30	1992	13:45		1207-55911	RT
12	11	10	5	1992	13:30		877-76323	RT
13	12	10	8	1992	13:45		877-76324	RT
14	13	10	9	1992	12:30		877-76325	RT
15	14	10	10	1992	11:05		1207-55917	RT
16	15	10	11	1992	11:00		1207-55912	RT
17	16	10	11	1992	11:45		1207-55913	RT
18	17	10	11	1992	12:40		877-76326	RT
19								

This article won't spend much time on the M language. Just know that the language is the driving force behind the Power BI Desktop transformations in Query Editor. Articles later in this series will dig into the language in more detail so you can work with it directly.

Every transformation you apply to a dataset is listed in the *Applied Steps* section, and each step is associated with an M statement that builds on the preceding step. You can use the steps and their associated M statements to work with the dataset in various ways. For example, if you select a step in the *Applied Steps* section, Query Editor will display the dataset as it existed when the step was applied. You can also modify steps, delete steps, inject steps in between others, or move steps up or down. In fact, the *Applied Steps* section is one of the most powerful tools you have for working with Power BI Desktop datasets. Be aware, however, that it's very easy to introduce an error into the steps that turns all your work into a big mess.

With that in mind, you'll next provide column names for the dataset. You might have noticed from the screenshots—or when you imported the data for yourself—that the first row of data contains the original column names. This is why all columns but the first were assigned the **text** data type, even if most values were numbers. The first column was assigned the **Int64** data type because it was the index for the original dataset and did not require a column name.

To convert the first row into columns names, click the *Use First Row as Headers* button on the *Home* ribbon. Power BI Desktop will promote the row and add a *Promoted Headers* step to the *Applied Steps* section, as well as a *Changed Type* step, which converts the data types of any fields that might be impacted by removing the column names as data values.

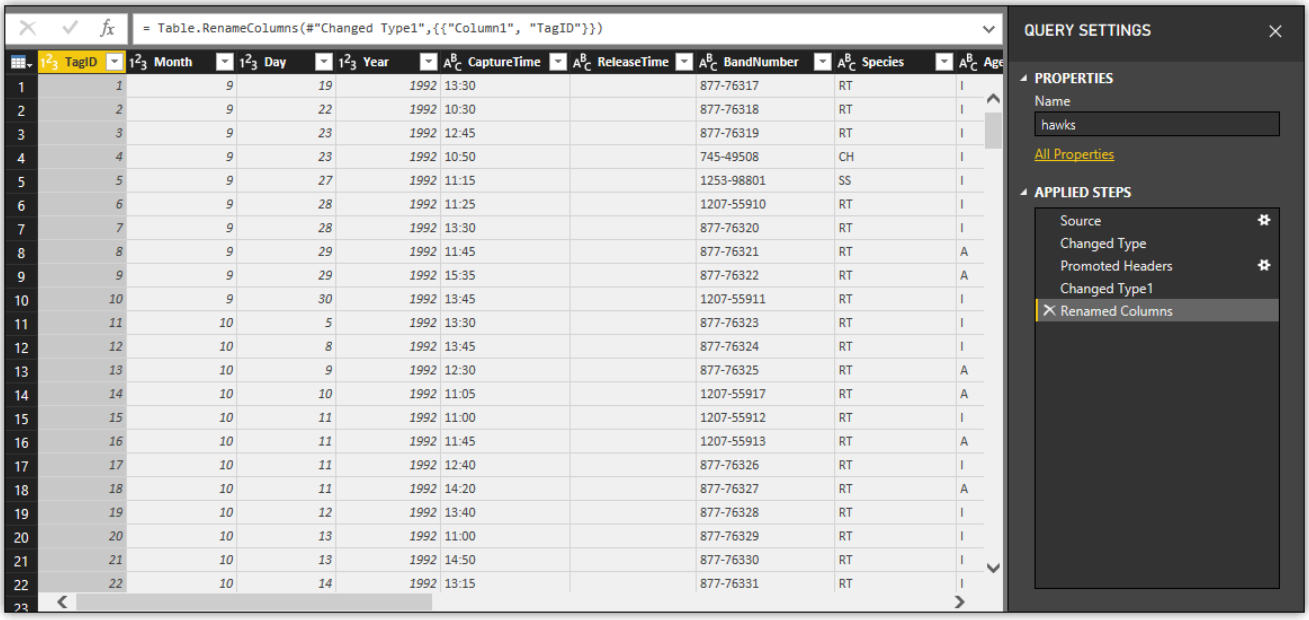
The following M statement is associated with the *Changed Type* step added to the *hawks* dataset:

```
= Table.TransformColumnTypes(#"Promoted Headers",{{"Column1", Int64.Type}, {"Month", Int64.Type}, {"Day", Int64.Type}, {"Year", Int64.Type}, {"CaptureTime", type text}, {"ReleaseTime", type text}, {"BandNumber", type text}, {"Species", type text}, {"Age", type text}, {"Sex", type text}, {"Wing", type number}, {"Weight", type text}, {"Culmen", type text}, {"Hallux", type text}, {"Tail", Int64.Type}, {"StandardTail", type text}, {"Tarsus", type text}, {"WingPitFat", type text}, {"KeelFat", type text}, {"Crop", type text}})
```

Notice that, in addition to the first column, several other columns have been converted to the **Int64** data type. However, the *Wing* column has been converted to the **number** type because it contains at least one decimal value. Other columns retained the **text** data type because they contained only character values or contained a mix of types.

It should be noted that the *Changed Type* step just added to the *hawks* data set is actually named *Changed Type1*. If a step is added that is of the same type as a previous step, Query Editor tags a numerical value to onto the new step to differentiate its name from the earlier step.

To complete the process of promoting the first row to headers, you might need to rename one or more columns. For example, because the *hawks* dataset did not include a name for the first column, change it from the auto-generated *column1* name to *TagID,* as shown in the following figure by right-clicking into the *column1* heading and selecting *Rename*.



	TagID	Month	Day	Year	CaptureTime	ReleaseTime	BandNumber	Species	Age
1	1	9	19	1992	13:30		877-76317	RT	I
2	2	9	22	1992	10:30		877-76318	RT	I
3	3	9	23	1992	12:45		877-76319	RT	I
4	4	9	23	1992	10:50		745-49508	CH	I
5	5	9	27	1992	11:15		1253-98801	SS	I
6	6	9	28	1992	11:25		1207-55910	RT	I
7	7	9	28	1992	13:30		877-76320	RT	I
8	8	9	29	1992	11:45		877-76321	RT	A
9	9	9	29	1992	15:35		877-76322	RT	A
10	10	9	30	1992	13:45		1207-55911	RT	I
11	11	10	5	1992	13:30		877-76323	RT	I
12	12	10	8	1992	13:45		877-76324	RT	I
13	13	10	9	1992	12:30		877-76325	RT	A
14	14	10	10	1992	11:05		1207-55917	RT	A
15	15	10	11	1992	11:00		1207-55912	RT	I
16	16	10	11	1992	11:45		1207-55913	RT	A
17	17	10	11	1992	12:40		877-76326	RT	I
18	18	10	11	1992	14:20		877-76327	RT	A
19	19	10	12	1992	13:40		877-76328	RT	I
20	20	10	13	1992	11:00		877-76329	RT	I
21	21	10	13	1992	14:50		877-76330	RT	I
22	22	10	14	1992	13:15		877-76331	RT	I

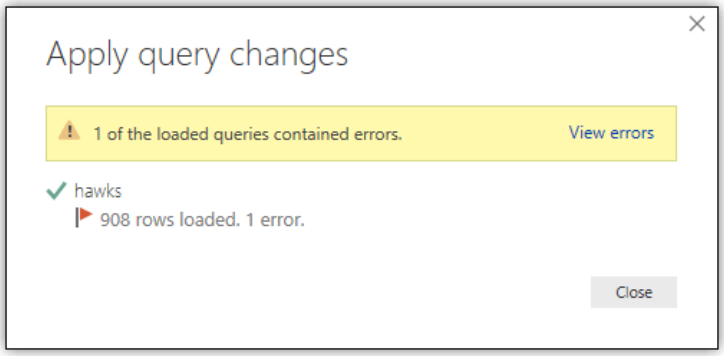
As you can see, the *Applied Steps* section now includes the *Renamed Columns* step. Also notice that each column name is preceded by a symbol that indicates the column's data type. If you click the symbol, a context menu will appear and display a list of symbols and their meanings. You can also change the data type from this menu.

Note, however, that the name of the data types listed here are not always the same as the names used in the M statements. For example, the **Int64** data type that appears in the M statements is listed as the *Whole Number* data type in the context menu.

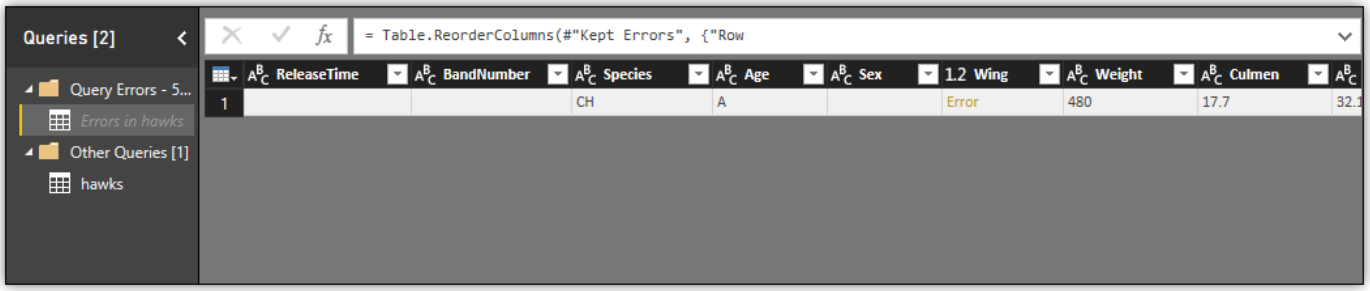
Addressing Errors

When working in Query Editor, you might run into errors when converting data, modifying applied steps, or taking other actions. Often you won't realize that there is an error until you try to apply any changes you've made. (You should be applying and saving your changes regularly when transforming data.)

To apply your changes in Query Editor, click the *Close & Apply* down arrow on the *Home* ribbon, and then click *Apply*. Any changes that you made since they last time you applied changes will be incorporated into the dataset, unless there is an error. For example, when you apply the changes to the *hawks* dataset after promoting the headers, you will receive the message shown in the following figure.



When you click the *View errors* link, Query Editor will isolate the row that contains the error, as shown in the following figure. The *TagID* value for this row is *263*. After viewing the error row, you should see that the *Wing* column contains an *NA* value, although the column is configured with the **number** data type, which is why you received the error.



Click hawks on the left to exit out of the *Errors* screen and get back to the Query Editor.

To address an error in Query Editor, you can replace or remove the error or change the column's data type:

- To change the column's type, click the type icon next to the column name, and then click *Text*.
- To replace the error, right-click the column header, click *Replace Errors,* type a replacement value, and then click *OK*.
- To filter out the row with the error, click the down arrow next to the *TagID* header, search for the *263,* and then clear the checkbox associated with this ID.
- To remove all rows that contain errors, click the table icon in the top left corner of the data set, and then click *Remove Errors*. This is the approach I took for the *hawks* dataset. As a result, the *Removed Errors* step was added to the *Applied Steps* section.

NOTE: One thing interesting about the error is that it occurs as a result of promoting the first row to column headers and the subsequent type conversion that came with it. From what I can tell, Query Editor must use only a sample of the data when selecting the data type during this step. It does not appear to have anything to do with the Data Type Detection setting in the preview window. I tried all possible settings and the results were always the same.

Removing Columns

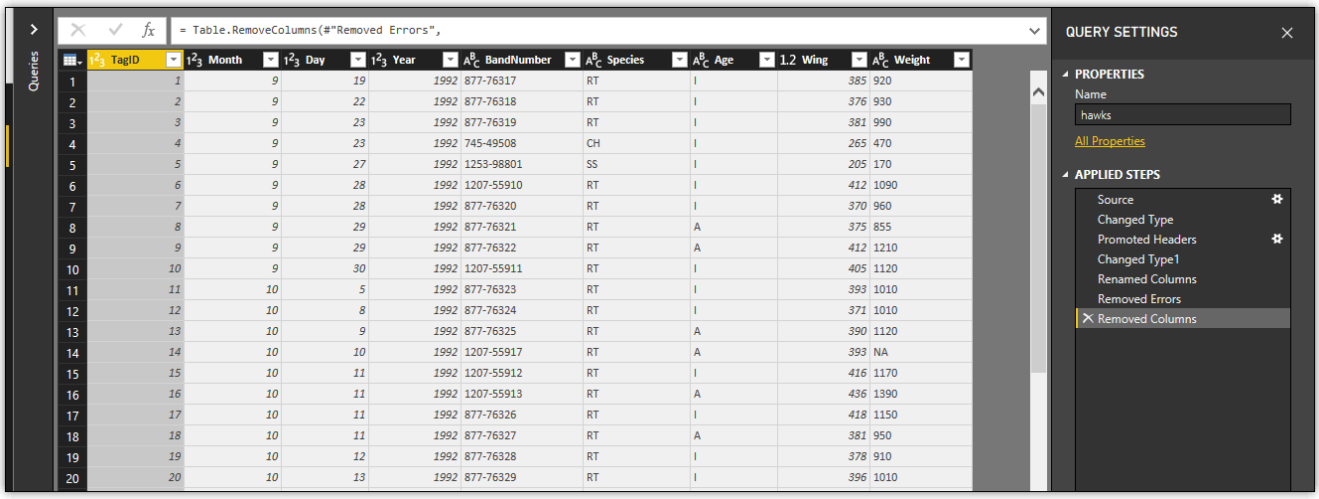
In some cases, the data you import includes columns that you don't need for your reports. Query Editor makes removing these columns a simple and quick process:

- To remove a single column, select the column directly in the displayed dataset, and then click the *Remove Columns* button on the *Home* ribbon, or you can instead right-click the column header and then click *Remove*.
- To remove multiple columns, select the first column, press and hold the *Control* key, select each additional column, and then click the *Remove Columns* button on the *Home* ribbon.

For the *hawks* dataset, remove the following columns because they are not relevant to the reports you might create or they contained numerous *NA* values:

- CaptureTime
- ReleaseTime
- Sex
- Culmen
- Hallux
- Tail
- StandardTail
- Tarsus
- WingPitFat
- KeelFat
- Crop

After removing the columns, Query Editor adds a single *Removed Columns* step to the *Applied Steps* section, as shown in the following figure.



	TagID	Month	Day	Year	BandNumber	Species	Age	Wing	Weight
1	1	9	19	1992	877-76317	RT	I	385	920
2	2	9	22	1992	877-76318	RT	I	376	930
3	3	9	23	1992	877-76319	RT	I	381	990
4	4	9	23	1992	745-49508	CH	I	265	470
5	5	9	27	1992	1253-98801	SS	I	205	170
6	6	9	28	1992	1207-55910	RT	I	412	1090
7	7	9	28	1992	877-76320	RT	I	370	960
8	8	9	29	1992	877-76321	RT	A	375	855
9	9	9	29	1992	877-76322	RT	A	412	1210
10	10	9	30	1992	1207-55911	RT	I	405	1120
11	11	10	5	1992	877-76323	RT	I	393	1010
12	12	10	8	1992	877-76324	RT	I	371	1010
13	13	10	9	1992	877-76325	RT	A	390	1120
14	14	10	10	1992	1207-55917	RT	A	393	NA
15	15	10	11	1992	1207-55912	RT	I	416	1170
16	16	10	11	1992	1207-55913	RT	A	436	1390
17	17	10	11	1992	877-76326	RT	I	418	1150
18	18	10	11	1992	877-76327	RT	A	381	950
19	19	10	12	1992	877-76328	RT	I	378	910
20	20	10	13	1992	877-76329	RT	I	396	1010

One of the advantages of having the *Removed Columns* step listed along with the other steps is that you can easily remove or modify the step if you later decide that one or more of the columns should not have been removed. In this case, you're relatively safe modifying this step because any steps that you might have added after deleting the columns cannot reference those columns because they're considered nonexistent. Close the Query Editor to go back to *Data* view.

Adding a Calculated Column

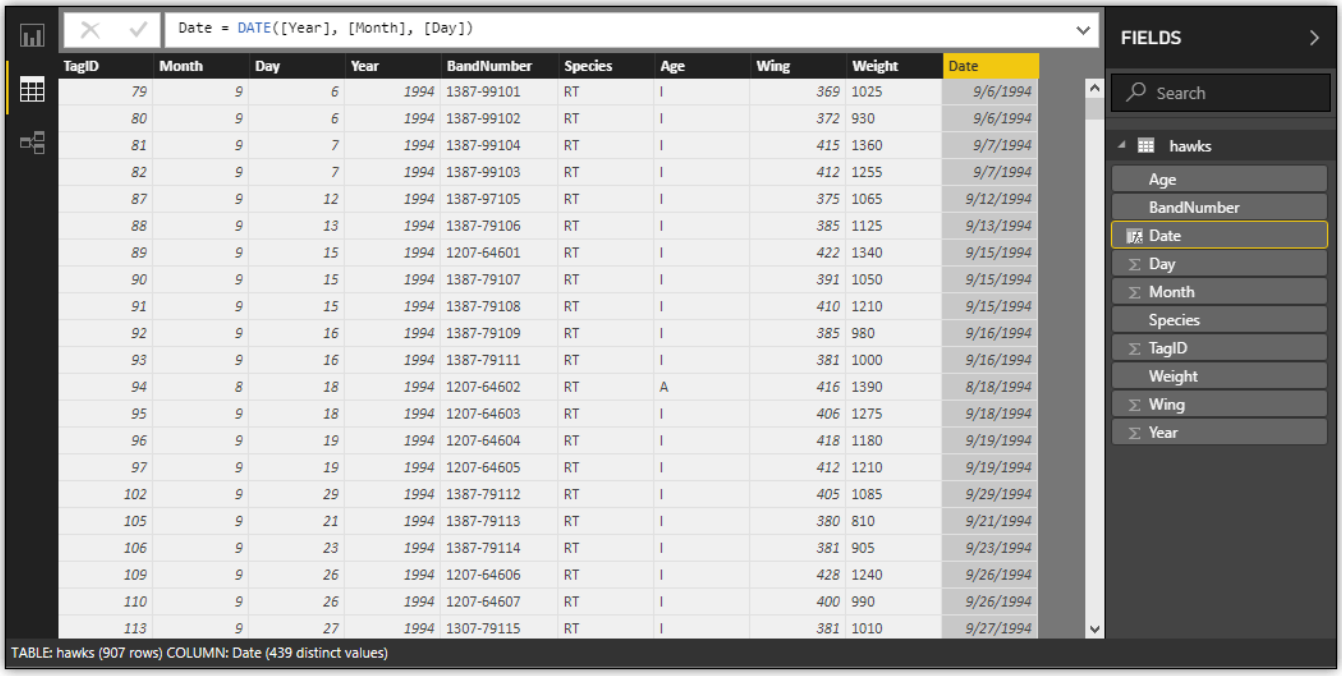
With Power BI Desktop, you can add calculated columns to a dataset that concatenates data or performs calculations. Power BI Desktop provides two methods for adding a calculated column. The first is to create the column in *Data* view, using the Data Analysis Expressions (DAX) language to define the column's logic.

To create a DAX-based column, click the *New Column* button on the *Home* ribbon in *Data* view, and then type the DAX expression in the formula bar at the top of the dataset. The new column should be selected when adding the expression. After you type the expression, click the checkmark to the left of the expression to verify the syntax and populate the new column.

For example, add a column named *Date* to the *hawks* dataset to provide a single value that shows the date when a hawk was tagged, using the following DAX expression:

```
Date = DATE([Year], [Month], [Day])
```

The expression uses the **DATE** function to create a date value based on the *Year, Month,* and *Day* columns. After you add the *Date* column, you can set the displayed format. Go to the *Modeling* ribbon in *Data* view, click the *Format* drop-down list, point to *Date Time,* and click *3/14/2001 M/d/yyyy*. The following figure shows the new column after I added it to the dataset, with the applied format.



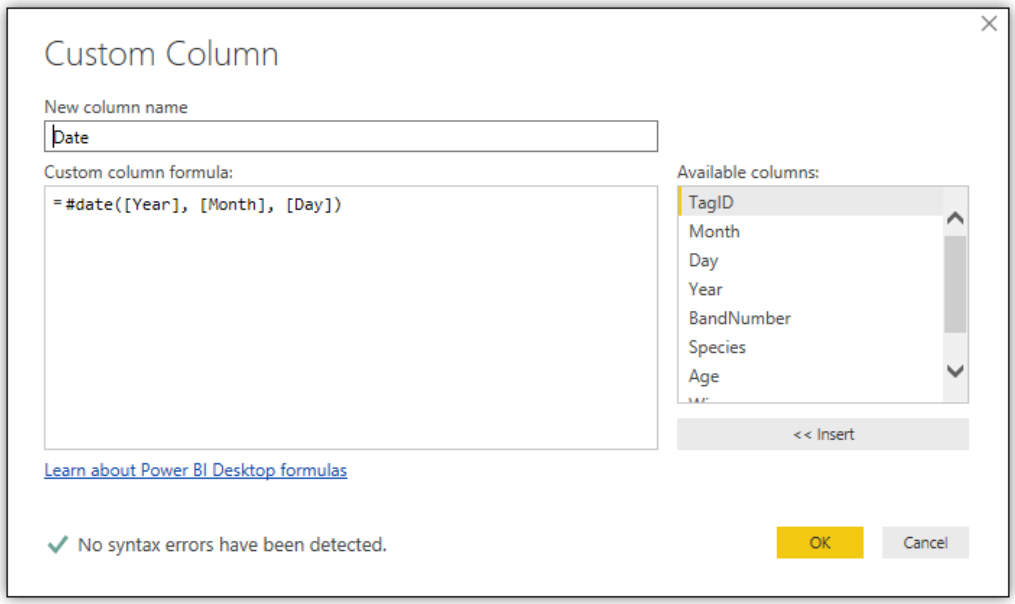
TagID	Month	Day	Year	BandNumber	Species	Age	Wing	Weight	Date
79	9	6	1994	1387-99101	RT	I	369	1025	9/6/1994
80	9	6	1994	1387-99102	RT	I	372	930	9/6/1994
81	9	7	1994	1387-99104	RT	I	415	1360	9/7/1994
82	9	7	1994	1387-99103	RT	I	412	1255	9/7/1994
87	9	12	1994	1387-97105	RT	I	375	1065	9/12/1994
88	9	13	1994	1387-79106	RT	I	385	1125	9/13/1994
89	9	15	1994	1207-64601	RT	I	422	1340	9/15/1994
90	9	15	1994	1387-79107	RT	I	391	1050	9/15/1994
91	9	15	1994	1387-79108	RT	I	410	1210	9/15/1994
92	9	16	1994	1387-79109	RT	I	385	980	9/16/1994
93	9	16	1994	1387-79111	RT	I	381	1000	9/16/1994
94	8	18	1994	1207-64602	RT	A	416	1390	8/18/1994
95	9	18	1994	1207-64603	RT	I	406	1275	9/18/1994
96	9	19	1994	1207-64604	RT	I	418	1180	9/19/1994
97	9	19	1994	1207-64605	RT	I	412	1210	9/19/1994
102	9	29	1994	1387-79112	RT	I	405	1085	9/29/1994
105	9	21	1994	1387-79113	RT	I	380	810	9/21/1994
106	9	23	1994	1387-79114	RT	I	381	905	9/23/1994
109	9	26	1994	1207-64606	RT	I	428	1240	9/26/1994
110	9	26	1994	1207-64607	RT	I	400	990	9/26/1994
113	9	27	1994	1307-79115	RT	I	381	1010	9/27/1994

Using DAX to create a column in *Data* view is quick and easy. However, this approach has some limitations. For example, the column is not available in Query Editor, so it cannot be used as part of another calculated column definition in Query Editor. In addition, if you delete a column referenced by a DAX expression, the values in the DAX-based column will show only errors.

The safest way to get around these limitations is to create your column in Query Editor. Delete the *Date* column you just added in the *Data* view. Click *Edit Queries* to reopen the Query Editor. Click the *Custom Column* button on the *Add Column* tab. When the *Custom Column* dialog box appears, type a name for the new column and the column's formula, which is an M expression that defines the column's logic. (Query Editor automatically adds the rest of the M syntax to create a complete statement.) For example, to create a column like the DAX-based column above, used the following M expression:

```
= #date([Year], [Month], [Day])
```

The statement uses the **#date** method to create a date value based on the *Year, Month,* and *Day* columns, similar to the DAX expression. The following figure shows what the M expression looks like when entered into the *Custom Column* dialog box.



When entering your M expression in the *Custom Column* dialog box, be sure that the green arrow is showing at the bottom of the dialog box to ensure that there are no syntax errors. Be aware, however, that you can have errors in your M statement without them showing up as syntax errors.

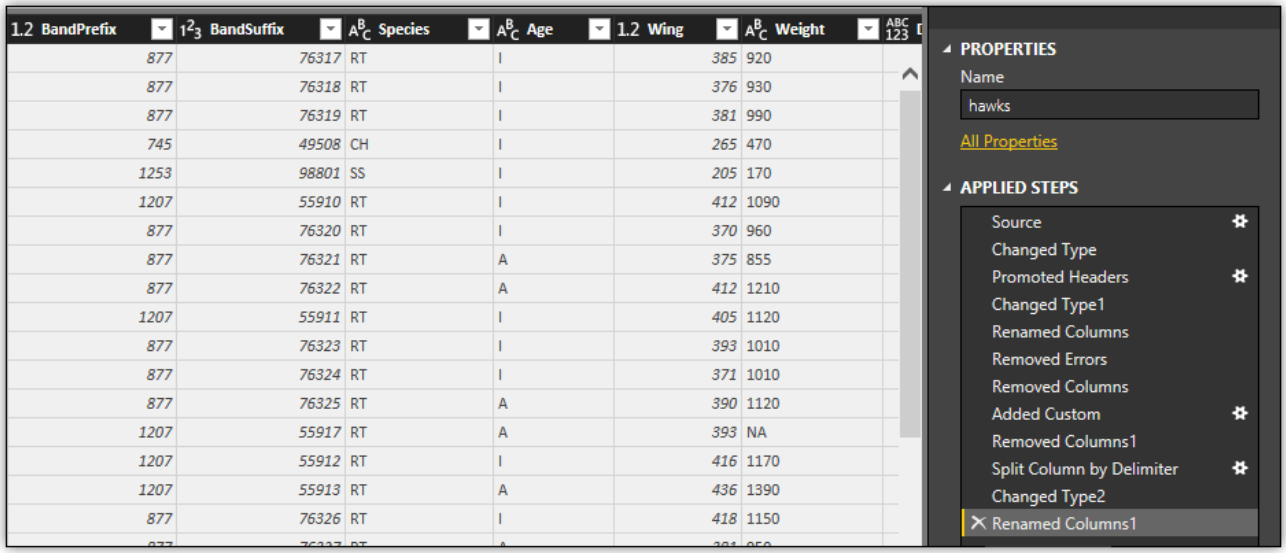
When you create a calculated column, Query Editor adds the *Added Custom* step to the *Applied Steps* section. You can then reference the new column in other calculated columns. You can also remove any of the columns referenced by the M expression. In this case, remove the *Month* and *Day* columns but keep the *Year* column in case you want it available for reports.

Splitting a Column

In Query Editor, you can split a column based on a specified value (delimiter) or by a specified number of characters. For example, the *hawks* dataset includes the *BandNumber* column, which is made up of two parts, separated by a hyphen. One of those parts might have special meaning, such as indicating the individual who tagged the hawk or the process used to tag the hawk. Splitting the column could make it easier to group the data by a particular entity.

To split a column by a delimiter, right-click the column's header, point to *Split Column*, and then click *By Delimiter*. In the *Split Column by Delimiter* dialog box, specify the delimiter and then click *OK*. Query Editor will split the column into two columns (removing the delimiter) and update the data types if necessary. You can then rename the columns if you do not want to use the autogenerated names.

The following figure shows the *hawks* dataset after splitting the *BandNumber* column and changing the names of the two new columns to *BandPrefix* and *BandSuffix*.



1.2 BandPrefix	1²3 BandSuffix	Species	Age	1.2 Wing	Weight
877	76317	RT	I	385	920
877	76318	RT	I	376	930
877	76319	RT	I	381	990
745	49508	CH	I	265	470
1253	98801	SS	I	205	170
1207	55910	RT	I	412	1090
877	76320	RT	I	370	960
877	76321	RT	A	375	855
877	76322	RT	A	412	1210
1207	55911	RT	I	405	1120
877	76323	RT	I	393	1010
877	76324	RT	I	371	1010
877	76325	RT	A	390	1120
1207	55917	RT	A	393	NA
1207	55912	RT	I	416	1170
1207	55913	RT	A	436	1390
877	76326	RT	I	418	1150

Notice that three steps have been added to the *Applied Steps* section: *Split Column by Delimiter, Changed Type2,* and *Renamed Columns1*.

Performing Additional Data Transformations

When you're preparing a dataset for reporting, you should evaluate the data types that have been assigned to each column to ensure they're what you need. Usually you'll want to hold off doing this until after you've added, removed, or split columns so you're not introducing unnecessary transformation steps. Evaluating the data types can also point to possible anomalies in the data.

For example, the *BandPrefix* column was assigned the **number** data type, which would have been expected to be the **Int64** data type. The same goes for the *Wing* column. In addition, the *Weight* column was assigned the **text** data type, and the *Date* column was assigned the **Any** data type. For this article, change the data types for the *BandPrefix, Wing*, and *Weight* columns to the **Int64** data type (whole number), and change the *Date* column to the **date** data type.

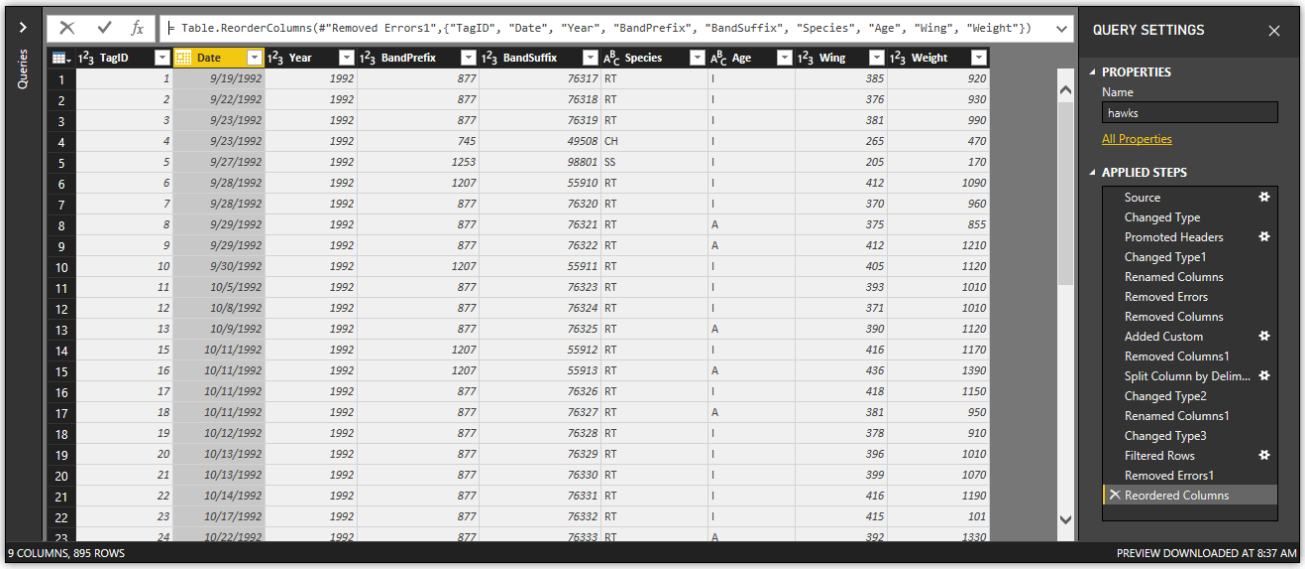
*NOTE: I could not find a clear answer for why Query Editor assigned the **number** type to the BandPrefix column and the **Int64** type to the BandSuffix column. The only numeric values I found in the BandPrefix column were whole numbers.*

However, when researching this issue, I discovered that both columns contained null values, so I removed those rows. To filter out rows with null values, click the down arrow in the applicable column header, and clear the checkbox associated with the null value.

The reason the *Wing* column was assigned the **number** data type was because it contained a decimal value. When changing the data type to **Int64**, Query Editor automatically rounded the value to the nearest whole number, which worked fine for the purposes of this article.

The process was not quite as simple for the *Weight* column. The reason it was assigned the **text** data type was because the column contained *NA* values, which were treated as errors when changing the column to the **Int64** data type. Remove these rows by using the *Remove Errors* option.

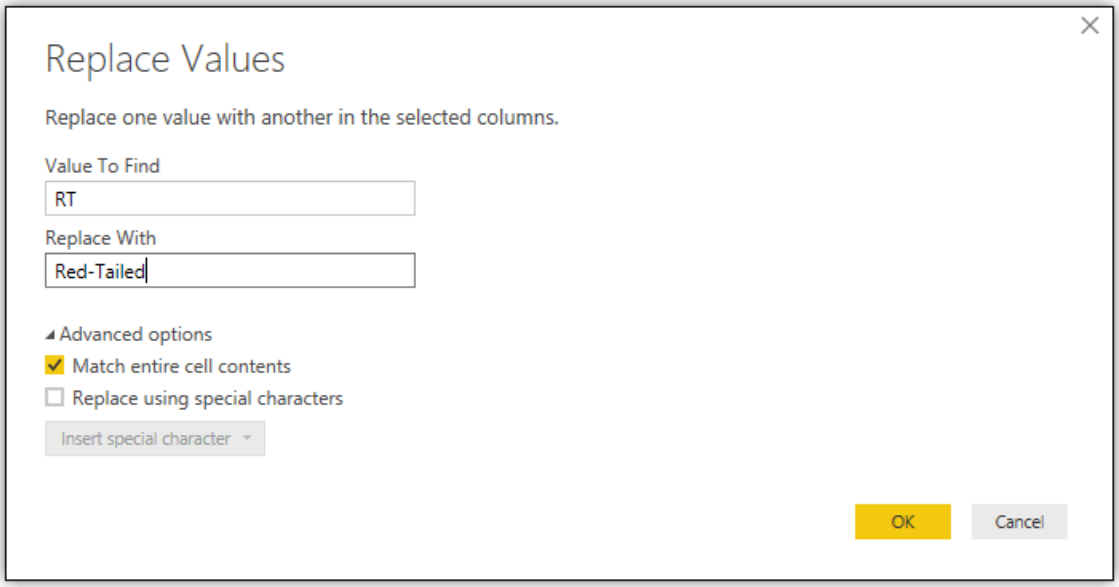
The next step is to move the *Date* column left within the dataset so it appears before the *Year* column. To move a column in Query Editor, simply drag the column header to the new position. Query Editor will add the *Reordered Columns* step to the *Applied Steps* section, as shown in the following figure. The figure also shows the other steps that were added when changing data types, filtered rows, and removed errors.



	TagID	Date	Year	BandPrefix	BandSuffix	Species	Age	Wing	Weight
1	1	9/19/1992	1992	877	76317	RT	I	385	920
2	2	9/22/1992	1992	877	76318	RT	I	376	930
3	3	9/23/1992	1992	877	76319	RT	I	381	990
4	4	9/23/1992	1992	745	49508	CH	I	265	470
5	5	9/27/1992	1992	1253	98801	SS	I	205	170
6	6	9/28/1992	1992	1207	55910	RT	I	412	1090
7	7	9/28/1992	1992	877	76320	RT	I	370	960
8	8	9/29/1992	1992	877	76321	RT	A	375	855
9	9	9/29/1992	1992	877	76322	RT	A	412	1210
10	10	9/30/1992	1992	1207	55911	RT	I	405	1120
11	11	10/5/1992	1992	877	76323	RT	I	393	1010
12	12	10/8/1992	1992	877	76324	RT	I	371	1010
13	13	10/9/1992	1992	877	76325	RT	A	390	1120
14	15	10/11/1992	1992	1207	55912	RT	I	416	1170
15	16	10/11/1992	1992	1207	55913	RT	A	436	1390
16	17	10/11/1992	1992	877	76326	RT	I	418	1150
17	18	10/11/1992	1992	877	76327	RT	A	381	950
18	19	10/12/1992	1992	877	76328	RT	I	378	910
19	20	10/13/1992	1992	877	76329	RT	I	396	1010
20	21	10/13/1992	1992	877	76330	RT	I	399	1070
21	22	10/14/1992	1992	877	76331	RT	I	416	1190
22	23	10/17/1992	1992	877	76332	RT	I	415	101
23	24	10/22/1992	1992	877	76333	RT	A	392	1330

The final step is to replace the values in the *Species* and *Age* columns to make them more readable. To replace a value, right-click the column header, and click *Replace Values*. In the *Replace Values* dialog box, type the original value and new value in the applicable text boxes. Next, click the *Advanced options* arrow, and select the *Match entire cell contents* checkbox. This is important to ensure that you don't replace part of another value.

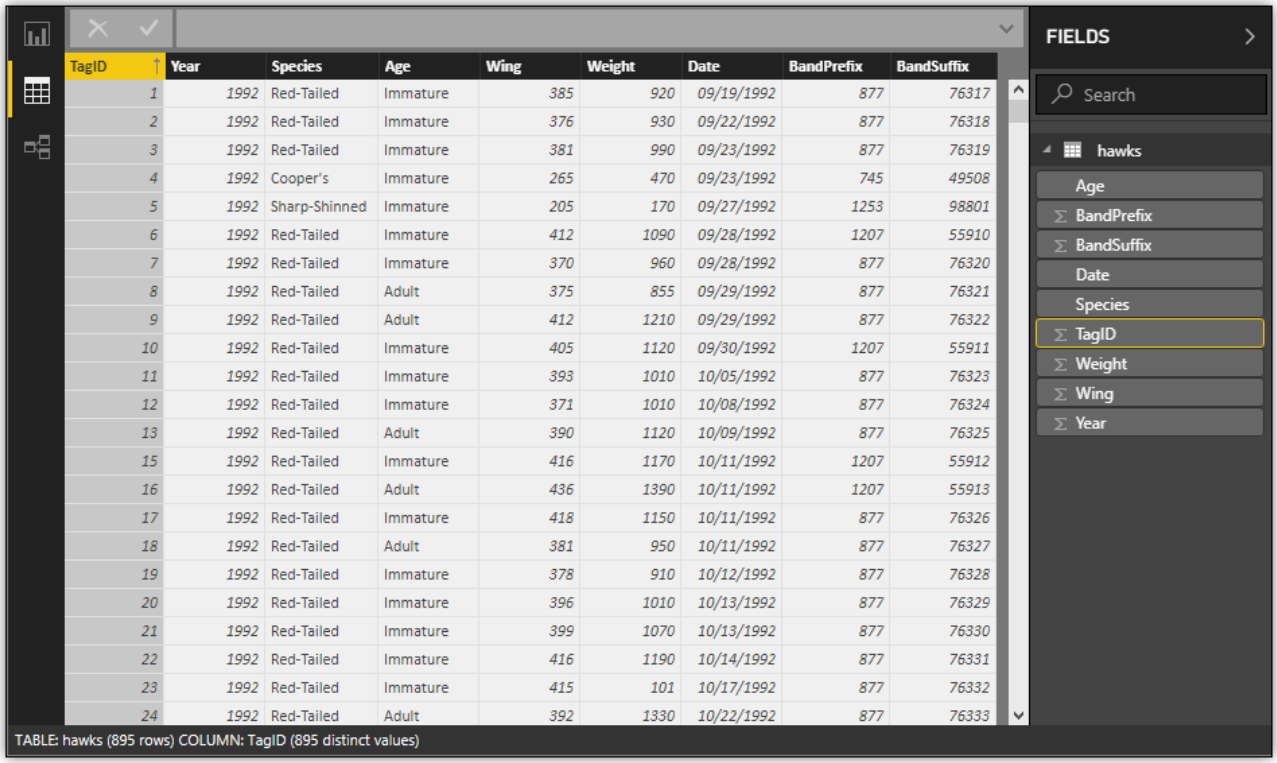
The first value to replace in the *hawks* data set is the *RT* value in the *Species* column. For the new value, used *Red-Tailed,* as shown in the following figure.



Also replace the following values:

- The *CH* value in the *Species* column with *Cooper's*.
- The *SS* value in the *Species* column with *Sharp-Shinned*.
- The *A* value in the *Age* column with *Adult*.
- The *I* value in the *Age* column with *Immature*.

The following figure shows the updated values in the *Species* and *Age* columns, as the dataset appears in the *Data* view of the main Power BI Desktop window.

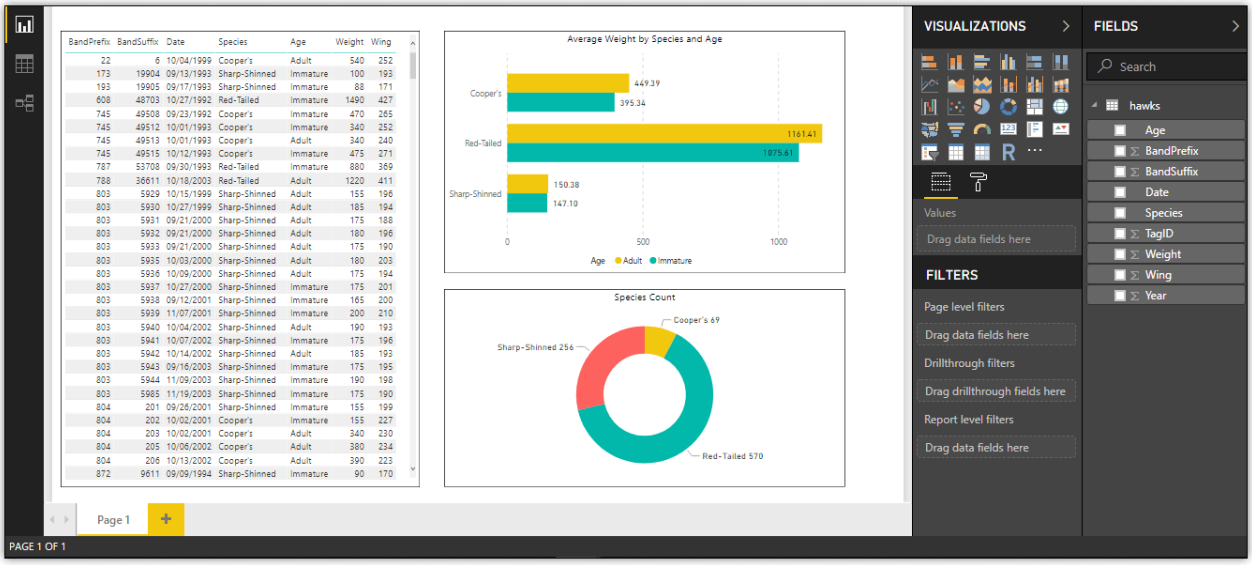


TagID	Year	Species	Age	Wing	Weight	Date	BandPrefix	BandSuffix
1	1992	Red-Tailed	Immature	385	920	09/19/1992	877	76317
2	1992	Red-Tailed	Immature	376	930	09/22/1992	877	76318
3	1992	Red-Tailed	Immature	381	990	09/23/1992	877	76319
4	1992	Cooper's	Immature	265	470	09/23/1992	745	49508
5	1992	Sharp-Shinned	Immature	205	170	09/27/1992	1253	98801
6	1992	Red-Tailed	Immature	412	1090	09/28/1992	1207	55910
7	1992	Red-Tailed	Immature	370	960	09/28/1992	877	76320
8	1992	Red-Tailed	Adult	375	855	09/29/1992	877	76321
9	1992	Red-Tailed	Adult	412	1210	09/29/1992	877	76322
10	1992	Red-Tailed	Immature	405	1120	09/30/1992	1207	55911
11	1992	Red-Tailed	Immature	393	1010	10/05/1992	877	76323
12	1992	Red-Tailed	Immature	371	1010	10/08/1992	877	76324
13	1992	Red-Tailed	Adult	390	1120	10/09/1992	877	76325
15	1992	Red-Tailed	Immature	416	1170	10/11/1992	1207	55912
16	1992	Red-Tailed	Adult	436	1390	10/11/1992	1207	55913
17	1992	Red-Tailed	Immature	418	1150	10/11/1992	877	76326
18	1992	Red-Tailed	Adult	381	950	10/11/1992	877	76327
19	1992	Red-Tailed	Immature	378	910	10/12/1992	877	76328
20	1992	Red-Tailed	Immature	396	1010	10/13/1992	877	76329
21	1992	Red-Tailed	Immature	399	1070	10/13/1992	877	76330
22	1992	Red-Tailed	Immature	416	1190	10/14/1992	877	76331
23	1992	Red-Tailed	Immature	415	101	10/17/1992	877	76332
24	1992	Red-Tailed	Adult	392	1330	10/22/1992	877	76333

Notice that the columns are not impacted by the reordering that was done in Query Editor. The newer columns—*Date, BandPrefix,* and *BandSuffix*—are included at the right side of the dataset, after the original columns. Also notice that in this case the *Date* column is configured in the format *mm/dd/yyyy,* but you might see the data values spelled out or displayed in another format. However, you can choose any format from the *Format* drop-down list on the *Modeling* ribbon.

Generating Reports

Once you get your datasets in the format you want them, you can begin to create reports and their visualizations. Just to show you a sample of what can be done, I created one report with three visualizations based on the *hawks* data set, as shown in the following figure.



The figure contains a one-page report as it appears in *Report* view in Power BI Desktop. The report includes a table, bar chart, and donut chart, all based on the *hawks* dataset. Although this article doesn't focus on reports, be aware that *Report* view includes a variety of options for creating and configuring different types of visualizations. Later in the series, you'll learn how to create reports and visualizations and publish them to the Power BI service.

Working with Power BI Desktop

There's much more you can do with Power BI Desktop and Query Editor than what was covered here. In addition to creating various types of visualizations, you can merge and append queries, group and aggregate data, pivot columns, define relationships, run R scripts, and carry out a number of other tasks. The best way to become familiar with these features is to experiment with them against different types of data. The better you understand how to work with Power BI Desktop, the more effectively you can use the rest of the Power BI tools, including the Power BI service, Power BI Report Server and the Power BI mobile apps.

Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3

R is a powerful language for manipulating data and creating visualizations. In this article in the series, Robert Sheldon demonstrates how R can be used within Power BI Desktop to import data and to create visualizations.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

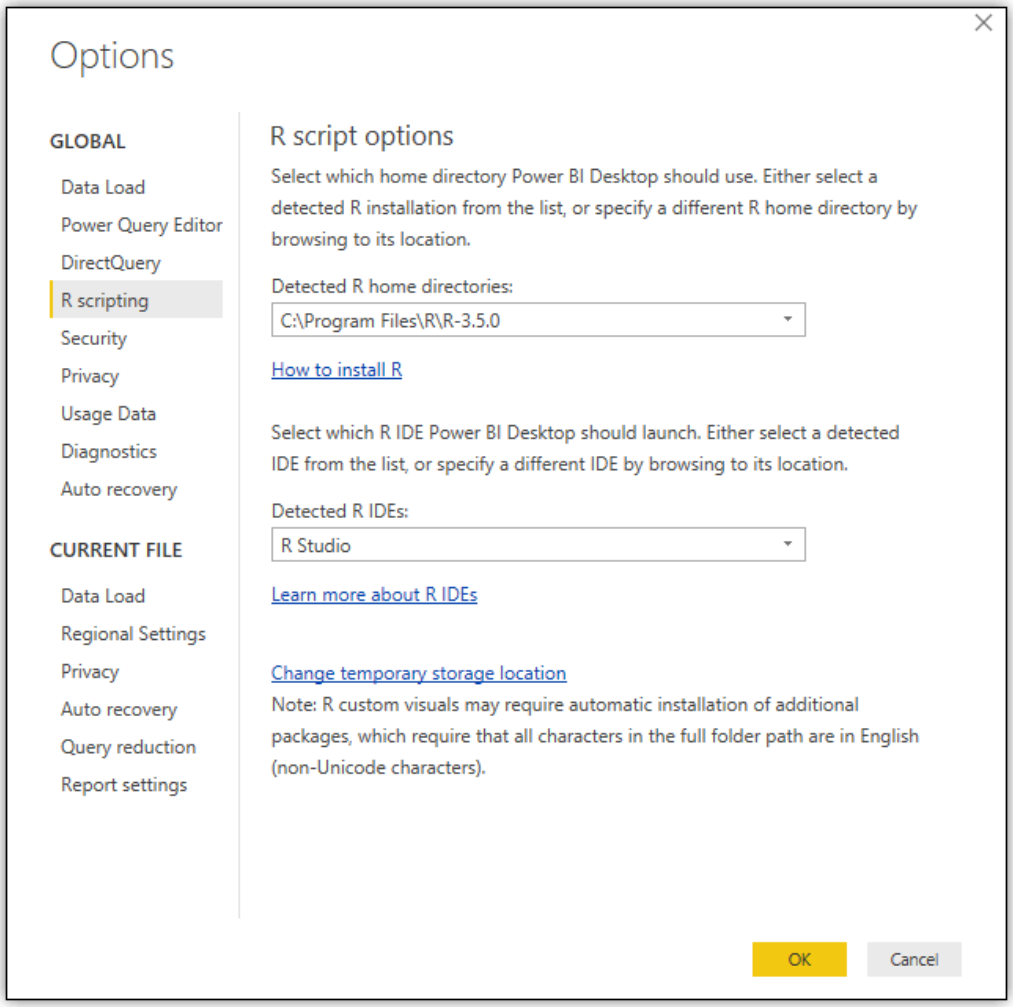
Power BI Desktop provides a powerful tool for creating reports you can publish to the Power BI service or save to Power BI Report Server. One of the most valuable features in Power BI Desktop is its integration with the R language. You can use R to import or transform data, as well as create visualizations. You can also import prebuilt R-based visualizations into Power BI Desktop, without needing to write or interact with R.

In this article, I explain how to work with R in Power BI Desktop, focusing on the how to perform R-specific tasks, as opposed to diving into the R language. For this reason, the R scripts in the examples are very rudimentary and meant only to demonstrate how you can utilize R in Power BI Desktop. Once you understand these concepts, you can then take advantage of R's extensive and robust capabilities to analyze, transform, and visualization data. If you're not familiar with R, a good place to start is with the R Project site.

When working with R in Power BI Desktop, R must be installed on the same Windows computer as Power BI Desktop. You can download and install an R distribution for free from a variety of locations, including the CRAN distribution available from the R Project site and the Microsoft R Open distribution available from the Revolution Analytics site. For the examples in this article, I used the CRAN distribution, version 3.5.0.

You'll also find it handy to have a separate R integrated development environment (IDE) installed on your system. This allows you to verify your R scripts before running them within Power BI Desktop. An IDE also makes it easier to install R packages that you might need when running R in Power BI Desktop. On my system, I installed the free version of RStudio, which is available from the RStudio site.

When you open Power BI Desktop, Power BI Desktop tries to locate an R installation on your system. You can verify whether the correct installation has been detected by opening the *Options* dialog box, available by going to *File* > *Options and settings* > *Options*. The following figure shows the *Options* dialog box with the *R scripting* section selected, as it appears on my system.



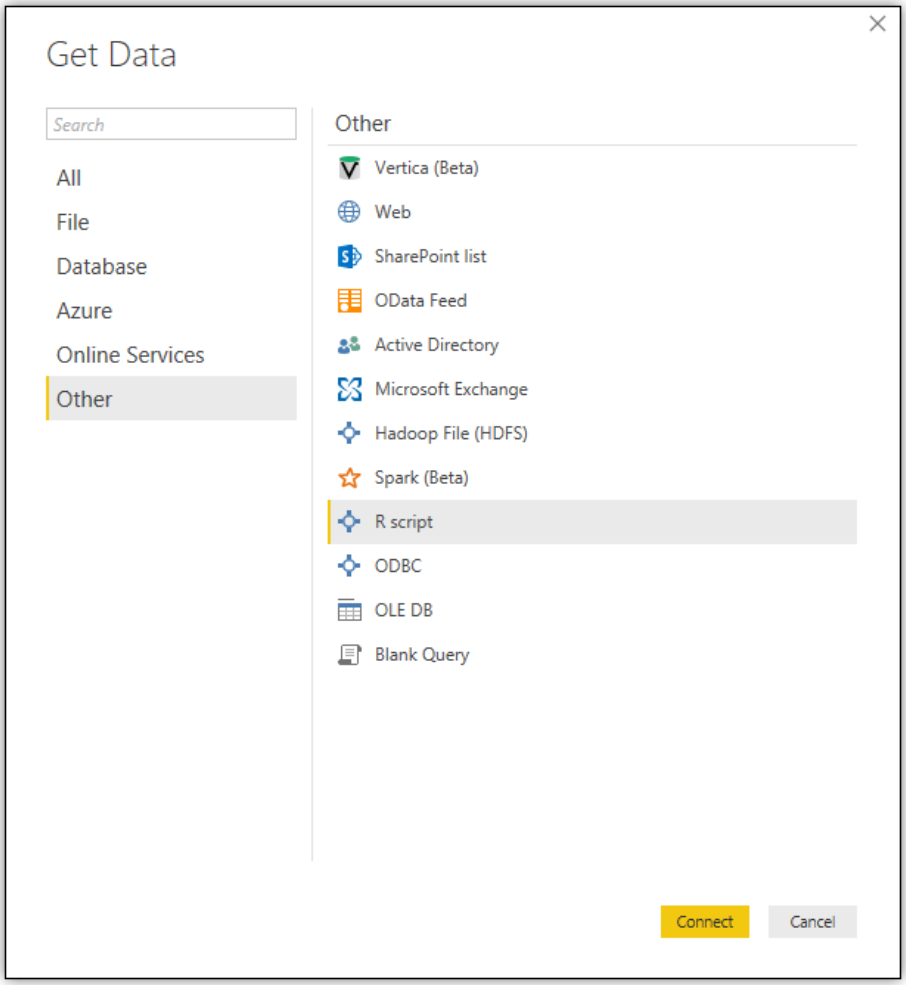
In my case, Power BI Desktop detected both the correct R installation and IDE. If you find the information to be incorrect or missing, you can provide a specific R installation, R IDE, or both.

Associating an IDE with Power BI Desktop applies only when creating R visualizations in *Report* view. By associating an IDE, you can launch the IDE from within Power BI Desktop and work on the visualization script there.

Using an R Script to Import Data

When using an R script to import data, the script must return at least one data frame that serves as the basis for your imported table. If the script returns multiple data frames, you can choose which ones to include as part of the import process. Power BI will create a table for each imported data frame. One word of warning, however. If a data frame contains a column configured with the **complex** or **vector** type, Power BI Desktop will replace the column's values with errors.

To use an R script to import data into Power BI Desktop, click the *Get Data* button on the *Home* ribbon. When the *Get Data* dialog box appears, navigate to the *R script* option in the *Other* category, as shown in the following figure.



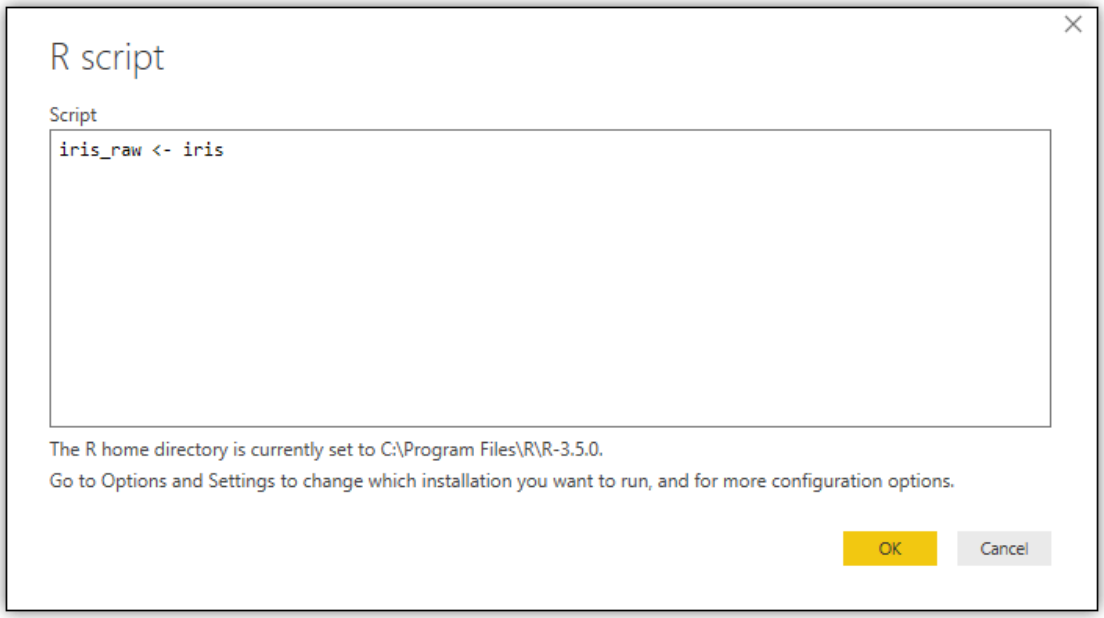
When you click the *Connect* button, Power BI Desktop launches the *R script* dialog box, where you can type or paste your R script. For the first example, use the following R script to retrieve data from the *iris* dataset (included with the CRAN distribution) and assign it to the *iris_raw* variable:

```
iris_raw <- iris
```

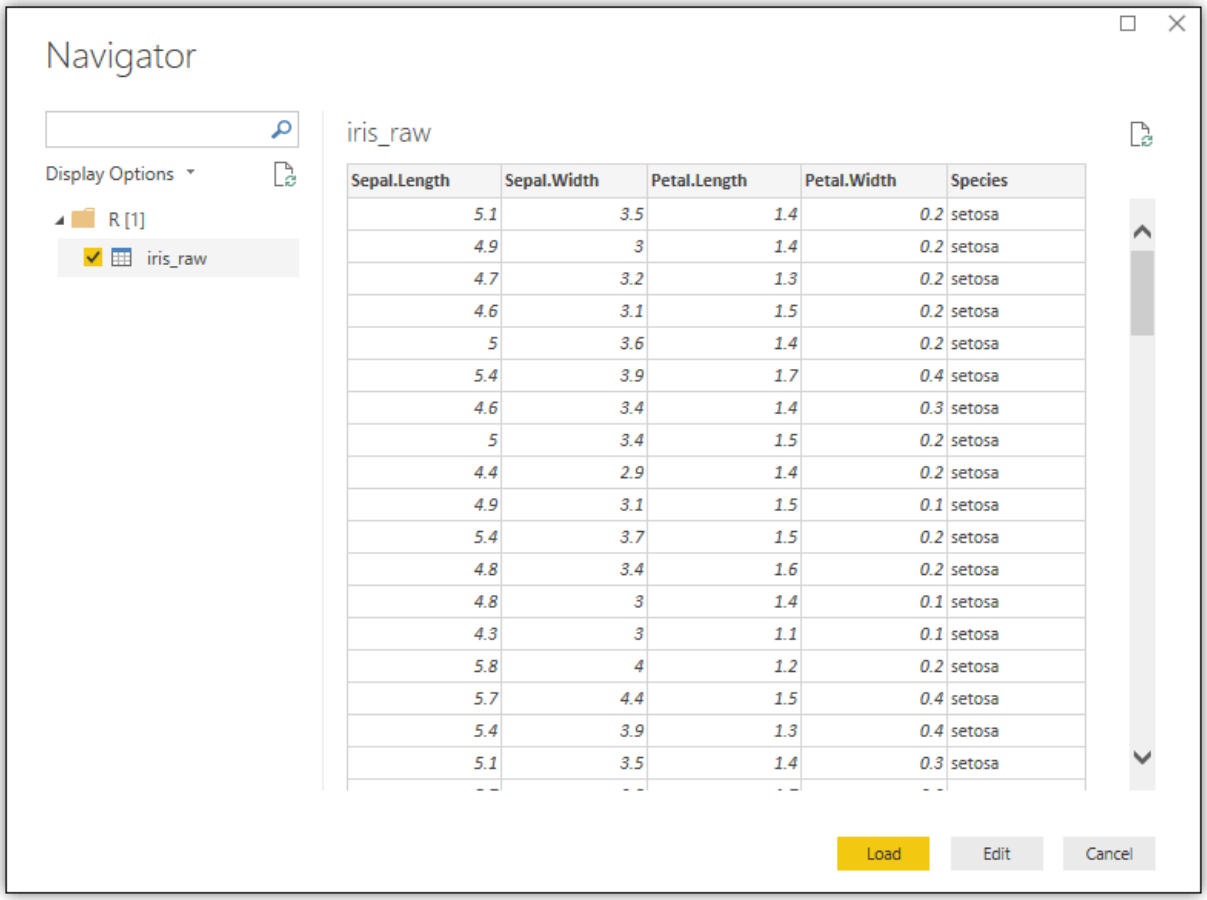
In Power BI Desktop, you must assign the dataset to a variable even if you don't modify the data frame in any way. If you were to enter only the name of the dataset, as you can do in an IDE, no data frames would be available to import into Power BI Desktop.

Before entering your script into the *R script* dialog box, you should test it in your IDE to ensure that it runs properly and returns the results you expect. If your script generates an error within Power BI Desktop, it can be very difficult to make sense of it. Plus, you have to start over with the import process.

Once you're sure that the script is ready to go, you can enter it into the *Script* textbox, as shown in the following figure.

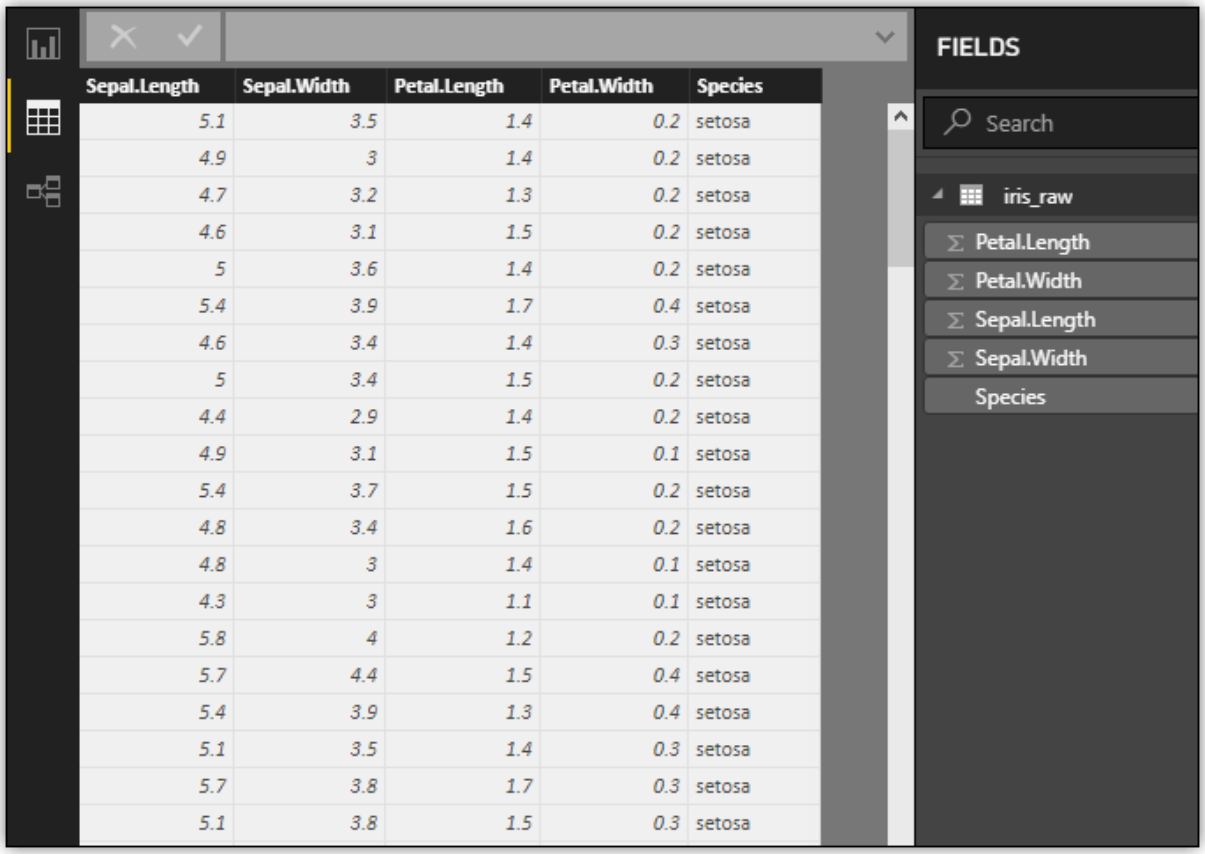


When you click *OK,* Power BI Desktop processes the script and then launches the *Navigator* dialog box, which lets you select which data frames to import we well as review sample data for each data frame. In this case, the R script returns only the **iris_raw** data frame, so it is the only one available, as shown in the following figure.



Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3	1.4	0.1	setosa
4.3	3	1.1	0.1	setosa
5.8	4	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa

The *Navigator* dialog box lists the data frames in the *Display Options* section, under the *R[1]* folder. You must select the checkbox associated with each data frame you want to import. Once you've made your selection, click *Load*. After Power BI Desktop loads the data, you can view the dataset in *Data* view, as shown in the following figure.



Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3	1.4	0.1	setosa
4.3	3	1.1	0.1	setosa
5.8	4	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

In some cases, you might want to import data from a file, rather than using one of the built-in datasets. For example, suppose you've copied the *iris* dataset to a CSV file, which you've saved to the *C:\DataFiles* folder on your local drive. You can easily pull the data into an R data frame using the following statement:

```
iris_csv <- read.csv(file="C:/DataFiles/iris.csv", header=TRUE, sep=",")
```

The statement uses the **read.csv** function to read the context of the *iris.csv* file. The **header** argument is set to **TRUE** to indicate that the first-row values should be created as headers (if that's what you want to do.) The **sep** argument indicates that a comma is used to separate the data values within the file. You can then use the same process described in the previous example to import the data from the **iris_csv** data frame.

The R language is very flexible due to the ability to install packages with additional functionality, and there are several that aid reporting and analytics. The **dplyr** and **data.table** packages provide useful functions (verbs) for working with data frames. The **ggplot2** is useful for visualizations. To use these in Power BI Desktop, they must be installed on your system. You can install these through your IDE, in this case R Studio. Launch R Studio and run these commands:

```
install.packages("dplyr")
install.packages("data.table")
install.packages("ggplot2")
```

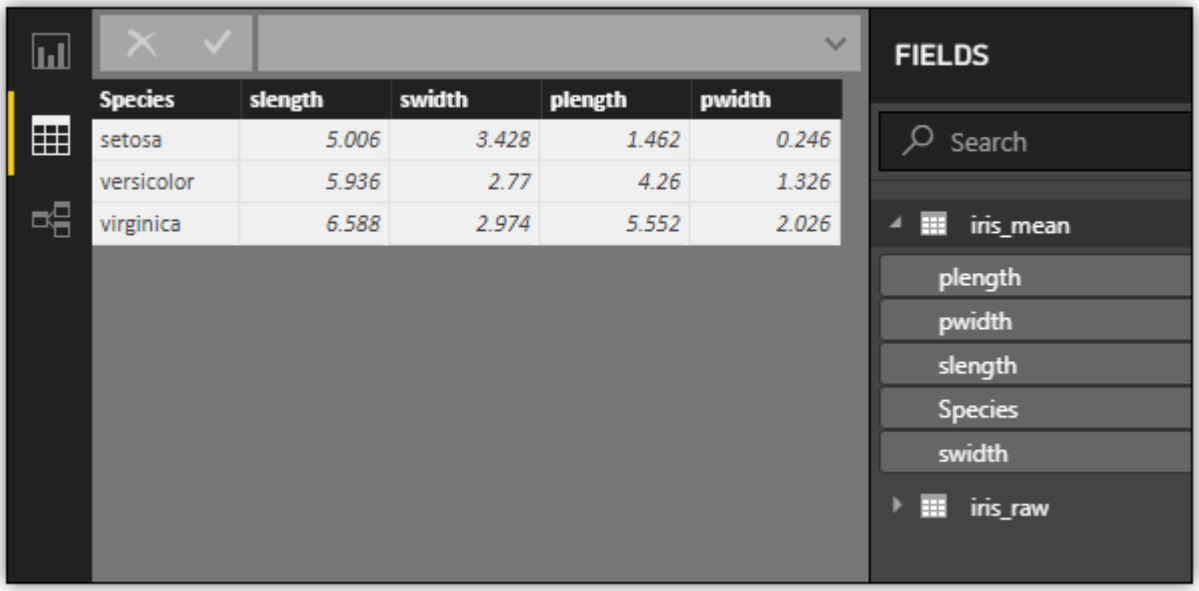
After the packages are installed, you can then use the **library** function within your R script to call that package when importing the data. This allows you to use the functions included in the package, such as **group_by** and **summarize**. To view a list of the packages that Power BI Desktop supports, see the help topic [R packages in the Power BI service](#).

One of the biggest benefits of being able to use R when importing data into Power BI Desktop is that you can manipulate data as part of the import process. For example, the following script uses the **summarize** and **group_by** functions available in the **dplyr** R package to group and aggregate the data before importing it:

```
library(dplyr)
iris_mean <- summarize(group_by(iris, Species),
  slength = mean(Sepal.Length), swidth = mean(Sepal.Width),
  plength = mean(Petal.Length), pwidth = mean(Petal.Width))
```

The **group_by** function prepares the data for use by another function, which in this case is **summarize** (also written as **summarise**). In this example, the **summarize** function is used in conjunction with the **mean** function to find the mean for each of the four measures, grouped according to the values in the *Species* column.

Returning now to the R statement above, notice that the aggregated data is saved to the **iris_mean** variable. This is the name that is assigned to the dataset when imported into Power BI Desktop, as shown in the following figure.



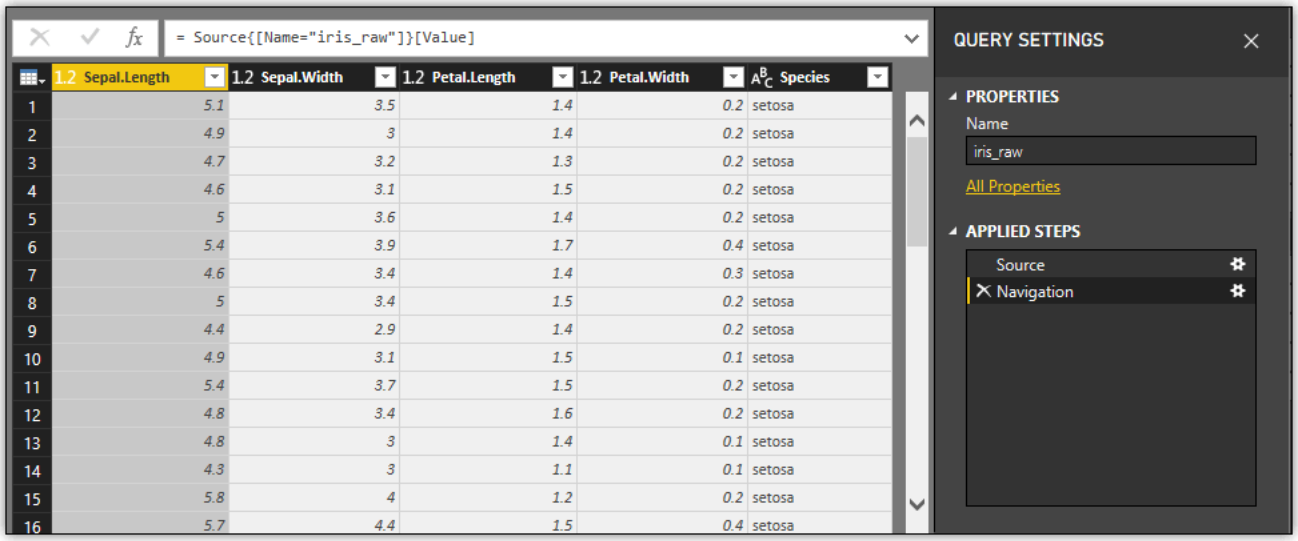
Of course, you can write far more complex R scripts than the one shown here, but this should give you an idea of how easy it is to use R to import data into Power BI Desktop. And the more adept you are with R, the more powerful this feature becomes.

Using an R Script to Transform Data

There might be times when you'll want to use R to manipulate a dataset already imported into Power BI Desktop. Query Editor includes the capacity to apply an R script to a dataset in order to transform the data. But before modifying the data, take a look at the following figure, which shows how the *iris_raw* dataset appears in Query Editor before applying any transformations.

Notice that the *Applied Steps* section includes two steps: *Source* and *Navigation*. In this figure, the *Source* step is selected and a small table is displayed in Query Editor's main pane. The table represents the initial import operation, with one row included for each returned data frame. In this case, because there is only the *iris_raw* data frame, the table includes only one row.

The *Table* value in the *Value* column represents the data associated with that data frame. Selecting this value results in the second step, *Navigation,* which is the actual imported data, as shown in the following figure. Whenever you use an R script to import data, Power BI Desktop adds both the *Source* and *Navigation* steps.

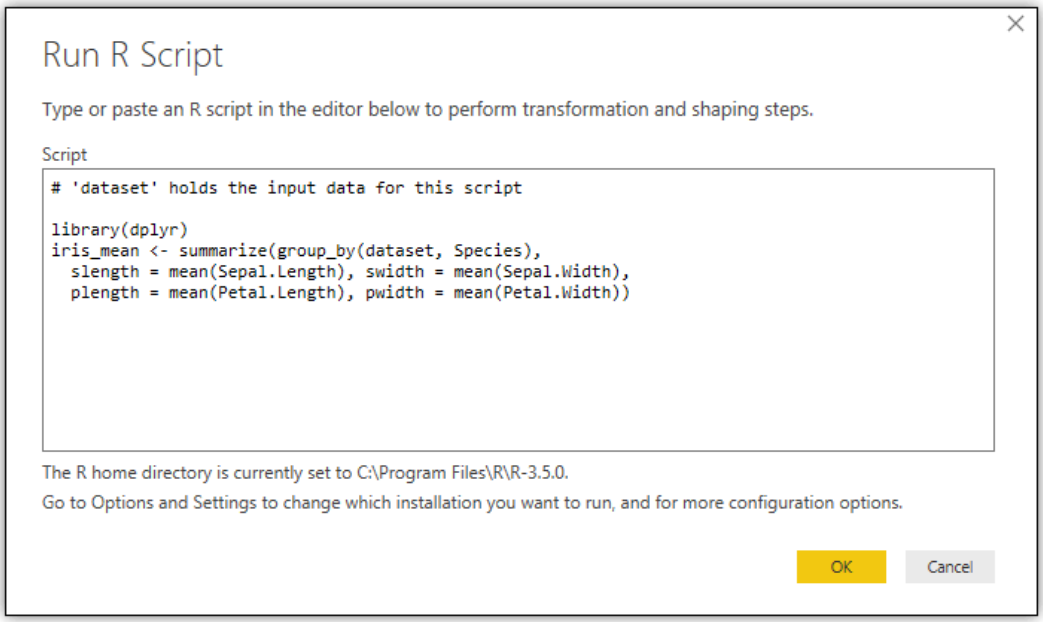


	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
8	5	3.4	1.5	0.2	setosa
9	4.4	2.9	1.4	0.2	setosa
10	4.9	3.1	1.5	0.1	setosa
11	5.4	3.7	1.5	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
13	4.8	3	1.4	0.1	setosa
14	4.3	3	1.1	0.1	setosa
15	5.8	4	1.2	0.2	setosa
16	5.7	4.4	1.5	0.4	setosa

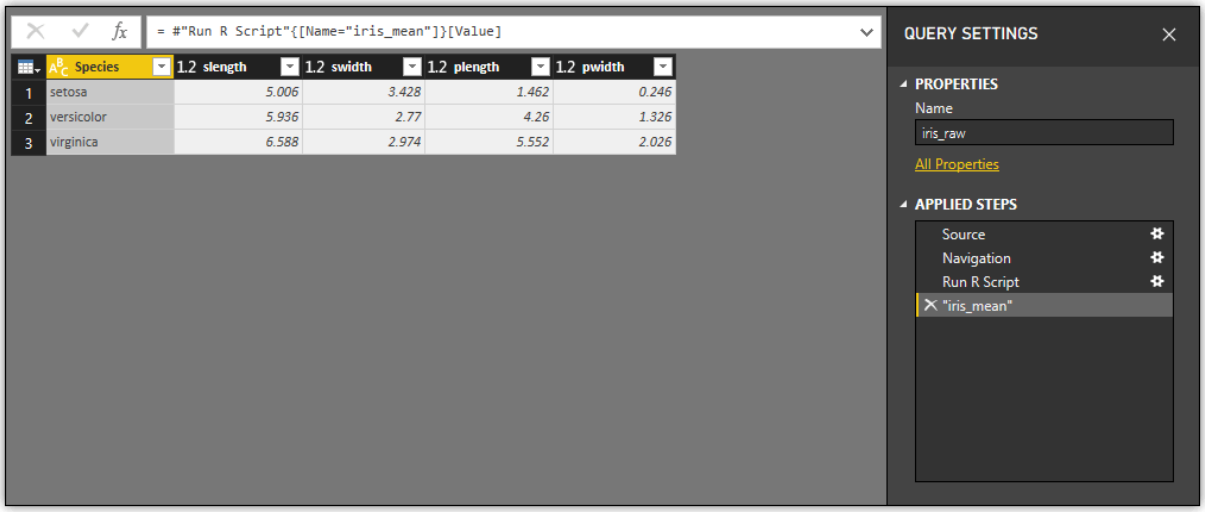
Now take a look at how to run an R script against the *iris_raw* dataset. To keep things simple, use the same aggregation logic used when importing the *iris_mean* dataset. The primary difference is that you must use the **dataset** variable when referencing the dataset, instead of specifying **iris**, as shown in the following script:

```
library(dplyr)
iris_mean <- summarize(group_by(dataset, Species),
  slength = mean(Sepal.Length), swidth = mean(Sepal.Width),
  plength = mean(Petal.Length), pwidth = mean(Petal.Width))
```

To run this or any R script in Query Editor, click the *Run R Script* button on the *Transform* ribbon. This launches the *Run R Script* dialog box, which includes the *Script* textbox, where you can type or paste your script. The following figure shows the *Run R Script* dialog box with the above R script included. Notice that Power BI Desktop adds a comment stating that the **dataset** variable holds the input data. The input data is the active dataset in Query Editor, which in this case is the *iris_raw* dataset.



After you enter the script and click *OK,* Query Editor adds two steps to the *Applied Steps* section: *Run R Script* and *"iris_mean"*. These two steps work just like the first two steps, *Source* and *Navigation*. The first one reflects the data frames returned by the script, and the second reflects the selected data frame, as shown in the following figure.



As with using R to import data, being able to run an R script against a dataset provides you with a powerful tool for working with the imported data, whether the data was imported from a database system, online service, or text file. After you've imported the data into Power BI Desktop, any dataset is fair game.

Using an R Script to Create Visualizations

Another great feature in Power BI Desktop is the ability to use R to create visualizations within *Report* view. For the most part, this process is as straightforward as using R in other ways, except for one important caveat: R automatically groups and summarizes the data, whether or not you want done, and you cannot override this behavior. Why Microsoft decided to take this approach is a bit of a mystery, especially since it can sometimes give you results you don't want.

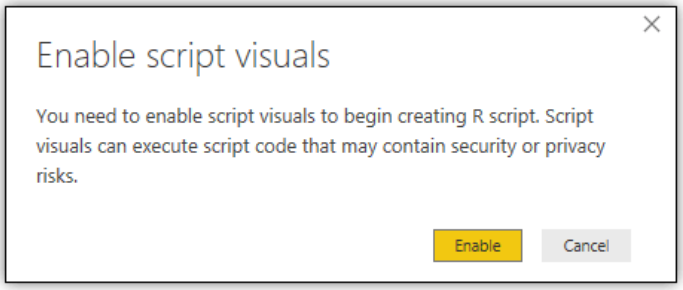
Fortunately, there is a workaround. You can add a column to your dataset that uniquely identifies each row, similar to an **IDENTITY** column in a SQL Server table. If you're using an R script to import the data, you can add the column as part of the import process. For example, the following R script adds an identifier column to the **iris** dataset based on the dataset's index (row names):

```
library(data.table)
iris_id <- iris
iris_id <- setDT(iris_id, keep.rownames=TRUE)[]
setnames(iris_id, 1, "id")
iris_id$id <- as.integer(iris_id$id)
```

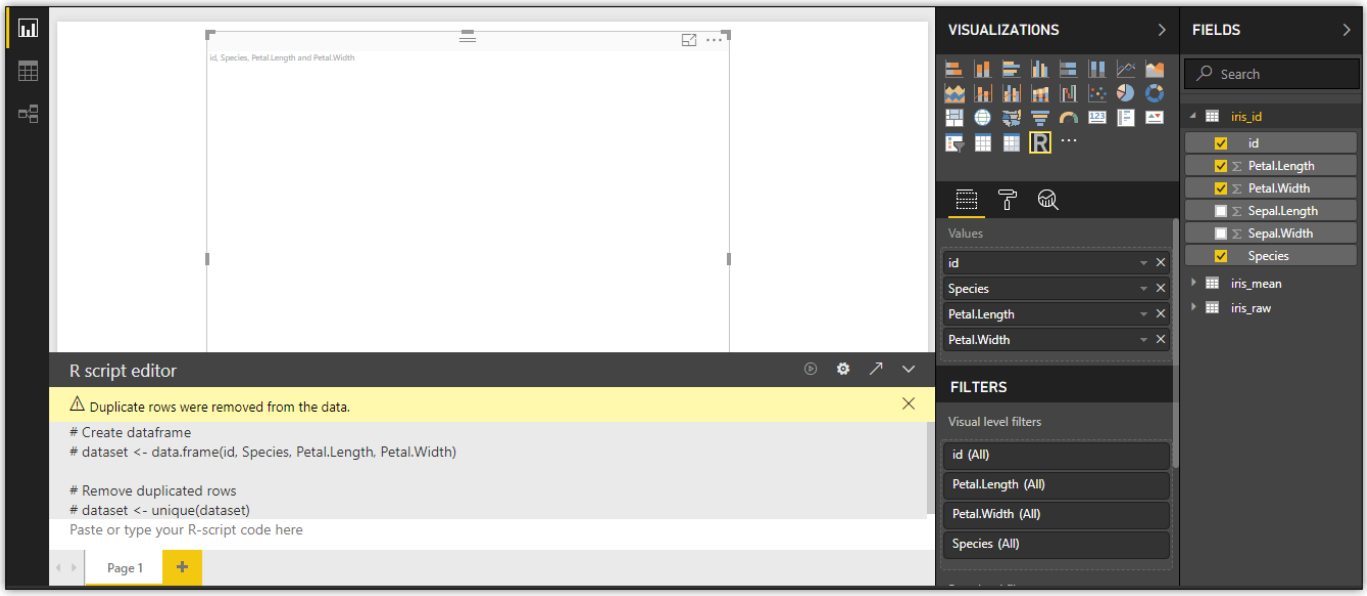
The script starts by calling the **data.table** package, which provides functions for working with data frame objects. (Be sure to install the package if you haven't already done so.) The **setDT** function, along with the **keep.rownames** argument, creates the new column based on the index values. Notice that you must first assign the **iris** dataset to the **iris_id** variable before using the **setDT** function. This is because the function makes changes directly to the dataset, which you cannot do with a built-in dataset such as **iris**.

After creating the column, you can then use the **setnames** function to change the name of the first column from the default (**rn**) to the new name (**id**). The final step is to change the column data type to **integer**.

With the *iris_id* dataset in place, you can use it to create a visualization based on an R script. To add an R-based visualization, go to *Report* view and click the *R* button on the *Visualizations* pane. The first time you do this, you'll be prompted to enable script visualizations, as shown in the following figure. Simply click the *Enable* button, and you'll be good to go.



When you click the *R* button on the *Visualizations* pane, Power BI Desktop adds a graphic placeholder to the report and opens the *R script editor* pane. Before you can do any scripting, however, you need to identify the dataset columns that you'll be using in the visualization. The easiest way to do this is to drag the columns from the *Fields* pane to the *Values* section of the *Visualizations* pane. Be sure to include the identifier column you created on the dataset. For this example, add the *id, Species, Petal.Length,* and *Petal.Width* columns, as shown in the following figure.



When you add the columns, Power BI Desktop inserts several comments in the *R script editor* pane. The first two comments indicate that a data frame named **dataset** has been created, based on the columns you added to the *Values* section of the *Visualizations* pane. You must use **dataset** to reference the source data in your R script.

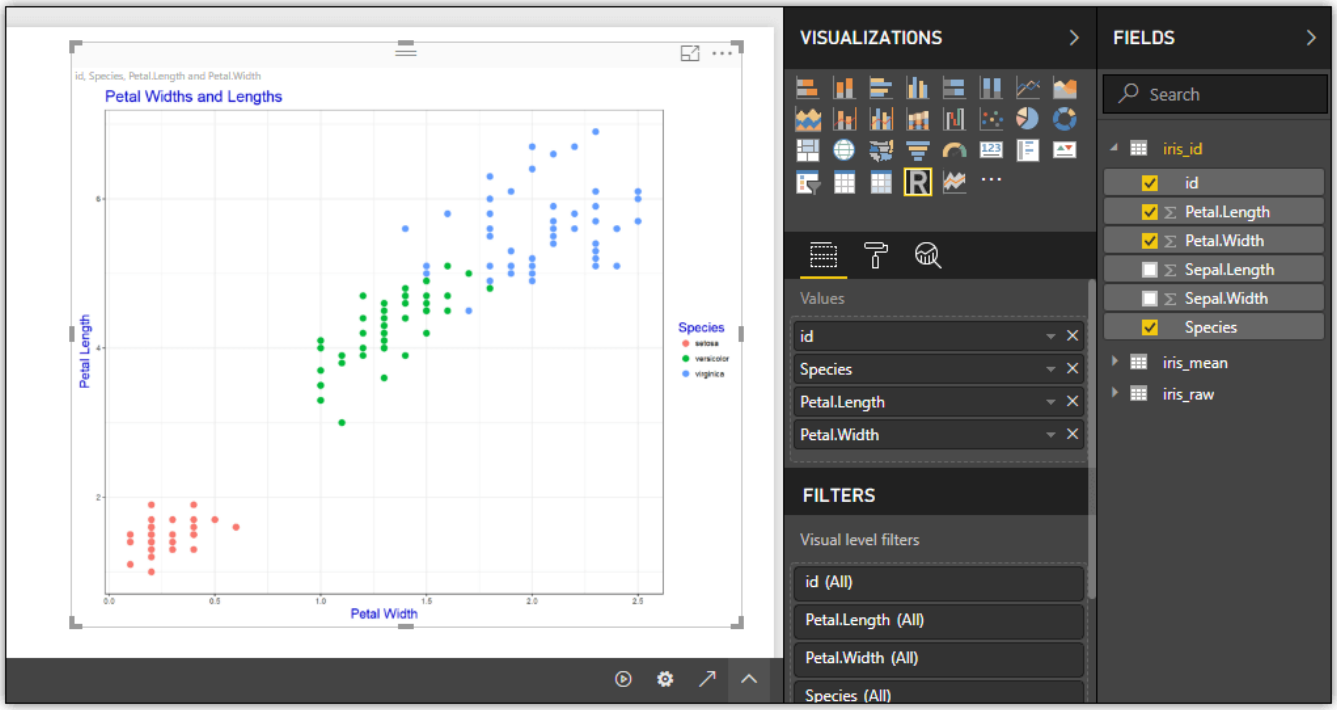
The second two comments, along with the warning message at the top of the pane, indicate that duplicate rows have been removed from the dataset, which is why you needed to add the identifier column.

Beneath the comments, you can type or paste your R script. For this example, use the following script to create a basic scatter plot:

```
library(ggplot2)
ggplot(data=dataset, aes(x=Petal.Width, y=Petal.Length)) +
  geom_point(aes(color=Species), size=3) +
  ggtitle("Petal Widths and Lengths") +
  labs(x="Petal Width", y="Petal Length") +
  theme_bw() +
  theme(title=element_text(size=15, color="blue3"))
```

The script uses the **ggplot** function in the **ggplot2** package (be sure to install the **ggplot2** package if you haven't already) to create a visualization with the specified colors and labels. Notice that the **Petal.Width** column is used for the X-axis, the **Petal.Length** column is used for the Y-axis, and the **Species** column serves as the basis for the plot colors.

After you define the R script, click the *Run script* button at the top of the *R script editor* pane. Power BI Desktop processes the script and displays the visualization in the placeholder added earlier, when you first clicked the *R* button. The following figure shows how the visualization appears on my system.



You can update an R-based visualization at any time by modifying the R script. After you modify the script, click the *Run script* button to update the visualization.

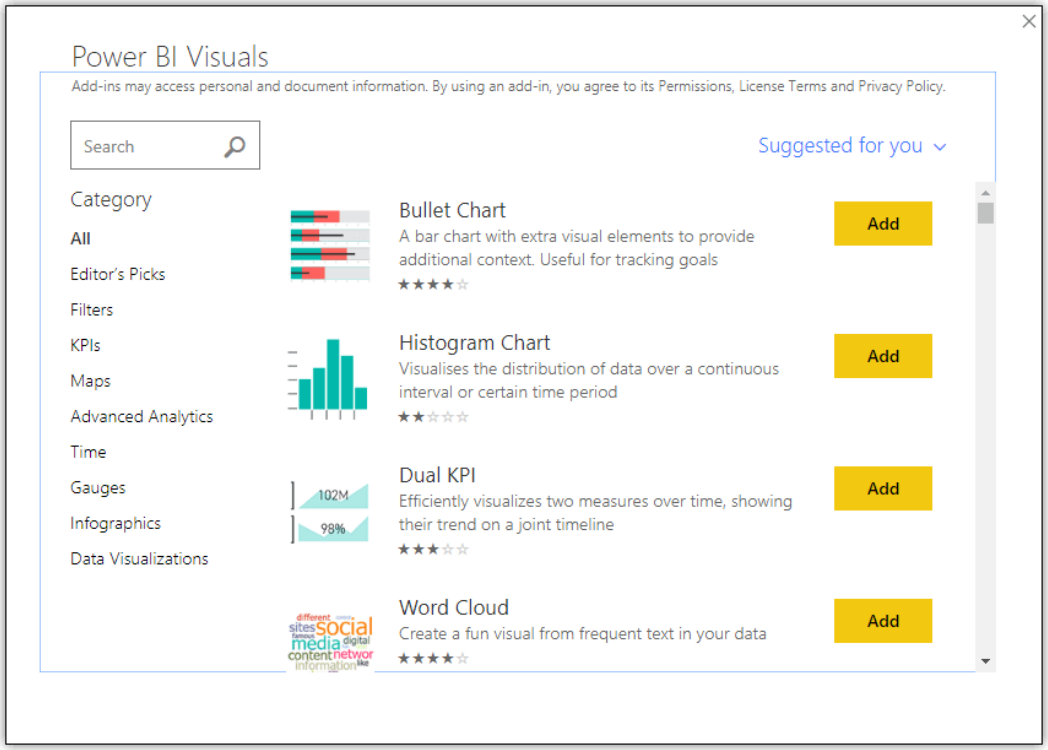
If you want to edit the code in your IDE, click *Edit script in external R IDE* at the top of the *R script editor* pane. This launches the IDE, which will display an R script that includes the code necessary to connect to the data source you're using for your visualization. The script will also include any code you already added to the visualization script in Power BI Desktop. You still have to copy-and-paste the bulk of your script back into Power BI Desktop, if you make any changes, but this at least saves you the trouble of trying to set up the data source in the IDE in order to test your script.

Something to keep in mind when adding R-based visualizations to your report is that the Power BI service comes with licensing restrictions. You cannot use R-based visualizations in the Power BI service unless you have a Power BI Pro license.

Importing R-based Custom Visuals

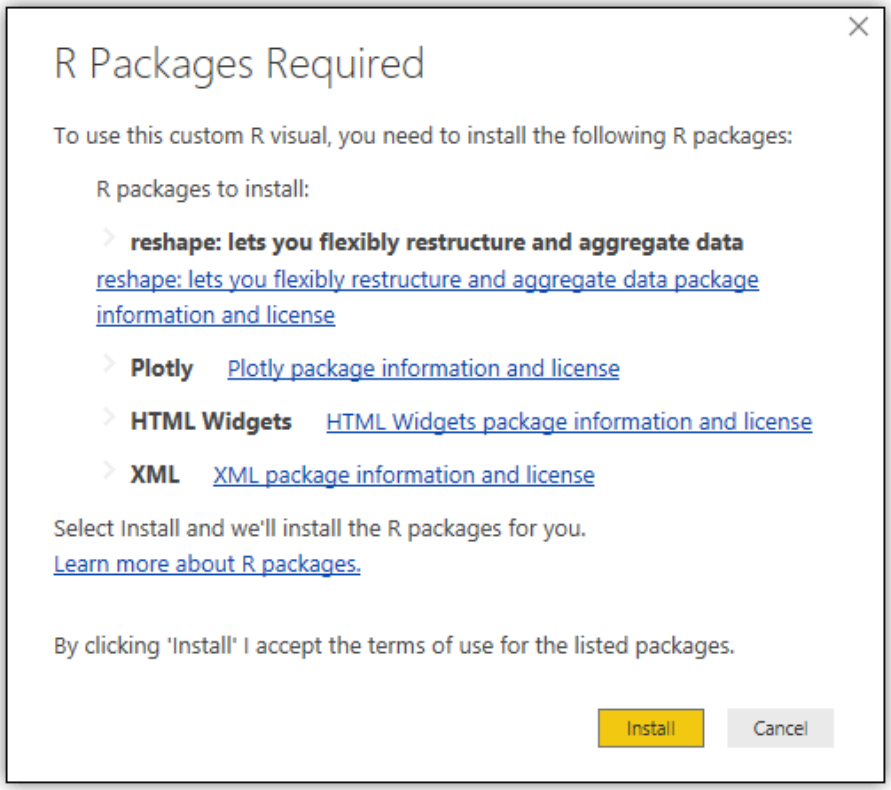
Another fun feature in Power BI Desktop is the ability to import predefined R-based visualizations into your workspace. The visualizations are available through Microsoft's AppSource gallery, which you can access directly through Power BI Desktop. You do not need to understand R syntax, nor do you need build or run R scripts.

To import a visualization, click the ellipses button on the *Visualizations* pane in *Report* view, and then click *Import from marketplace.* This launches the *Power BI Visuals* window, shown in the following figure.

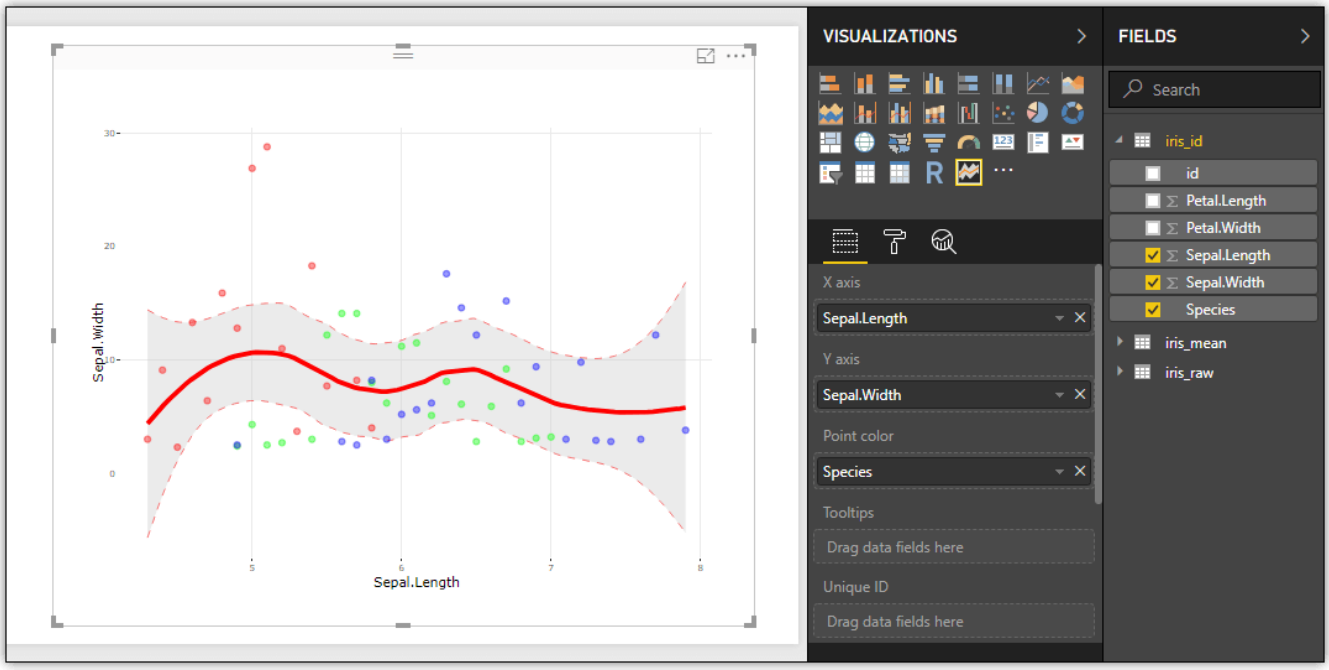


When you locate the visual you want to import, click *Add*. If the visualization requires additional R packages to be installed, the *R Packages Required* dialog box appears, listing the packages that need to be added. You can either click *Cancel* and install the packages manually, or you can click the *Install* button and let Power BI Desktop install the packages automatically.

For this example, select the *Spline chart* visualization. Type Spline in the *Search* window if it's hard to find. When it's added it to Power BI Desktop, the *R Packages Required* dialog box in the following figure will appear. To take the automatic installation route, click *Install*.



When you import a custom visualization, Power BI Desktop adds a button to the *Visualizations* pane specific to that visualization. You can then add the visualization to your report and configure just like you would any of the prebuilt visualizations. The following figure shows the *Spline chart* visualization. For the data, specify the *Species, Sepal.Length,* and *Sepal.Width* columns from the *iris_id* dataset, as shown in the following figure.



Microsoft provides a number of free and interesting custom visualizations in the AppSource gallery. It's certainly worth familiarizing yourself with what's out there so you can create reports that are even more engaging than you can achieve with the built-in visualizations. Best of all, Microsoft has made it exceptionally easy to pull the custom visualizations into Power BI Desktop, so you have little to lose by trying them out.

Making the Most of R in Power BI

The integration of the R language into Power BI Desktop provides a powerful tool for transforming and presenting business intelligence data. R is a comprehensive statistical computing and graphics language that is extensively implemented and has a large and active user community. In Power BI Desktop, you can use R to import and modify data as well as create a wide range of visualizations that provide in-depth insights into the data. Those already well versed in R should find that using R in Power BI Desktop is a simple and straightforward process. Even those less experienced with R could find it well worth the effort to try out the Power BI Desktop R features.

Power BI Introduction: Working with Parameters in Power BI Desktop —Part 4

Power BI allows you to use parameters to make your reports dynamic. In this article, Robert Sheldon demonstrates how parameters can be used to substitute connection information, query filters, or even calculations within the query.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Power BI Desktop, the downloadable application that supports the Power BI service, lets you define parameters that you can use in various ways when working with datasets. The parameters are easy to create and can be incorporated into the import process or later to refine a dataset or add dynamic elements. For example, you can create parameters that supply connection information to a data source or that provide predefined values for filtering data.

This article demonstrates how to create a set of parameters to use with data imported from a local SQL Server instance. The examples are based on data from the *AdventureWorks2017* database, but you can use an earlier version of the database or whatever database you choose, as long as the returned data follows a structure similar to what is used here. Just be sure to update the parameter values accordingly as you work through the article.

Adding Connection-Specific Parameters

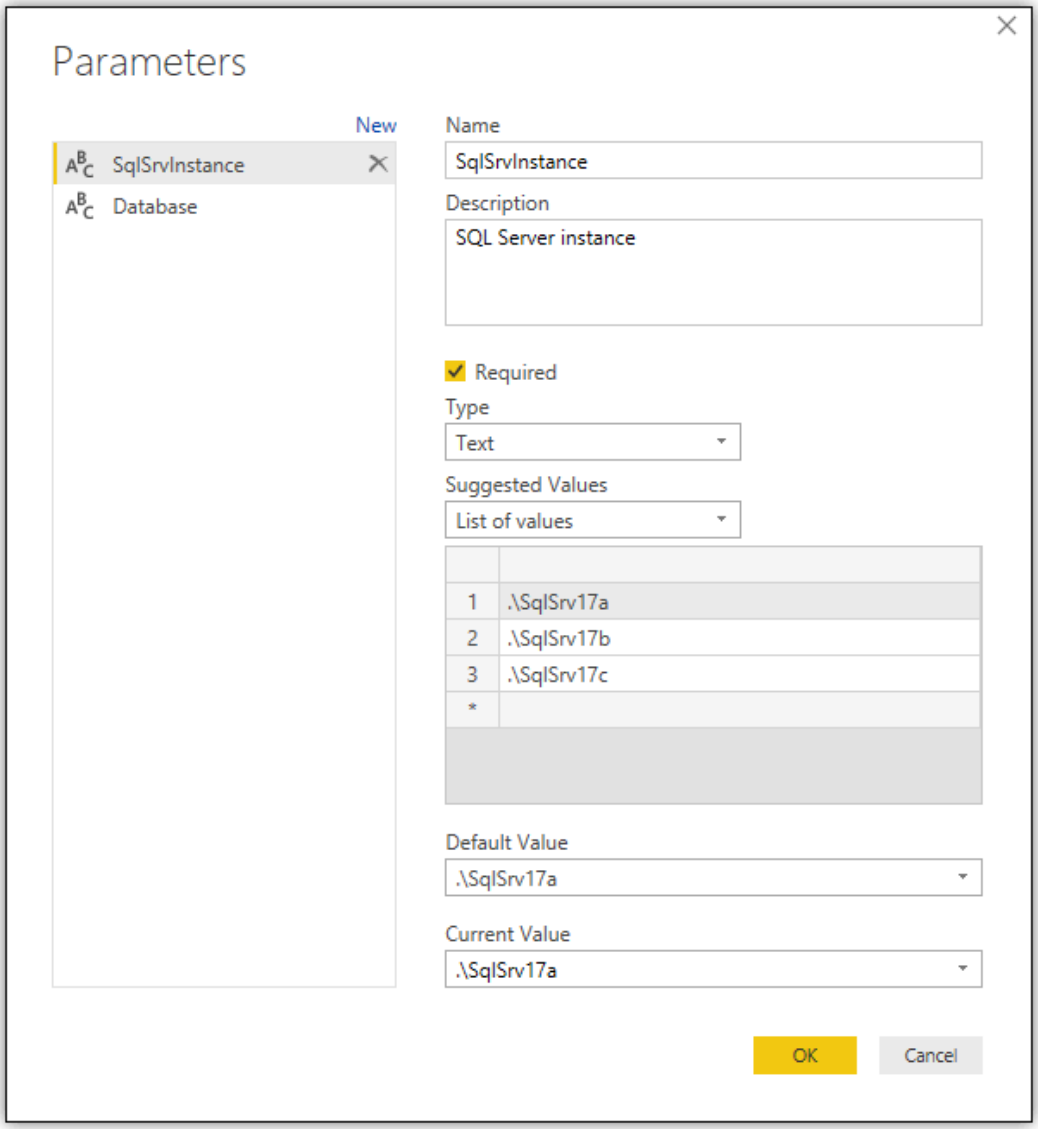
Some data source connections in Power BI Desktop—such as SharePoint, Salesforce Objects, and SQL Server—let you use parameters when defining the connection properties. For example, if retrieving data from SQL Server, you can use a parameter for the SQL Server instance and a parameter for the target database.

Because parameters are independent of any datasets, you can create them before adding a dataset or at any time after creating your datasets. However, you must define them and set their initial values in Query Editor. The parameters you create are listed in the *Queries* pane, where you can view and update their values, as well as reconfigure their settings.

In this article, you will create two connection parameters for retrieving data from a SQL Server database. The first parameter will include a list of SQL Server instances that could host the source data. (Only one instance needs to work in this case. The rest can be for demonstration purposes only.)

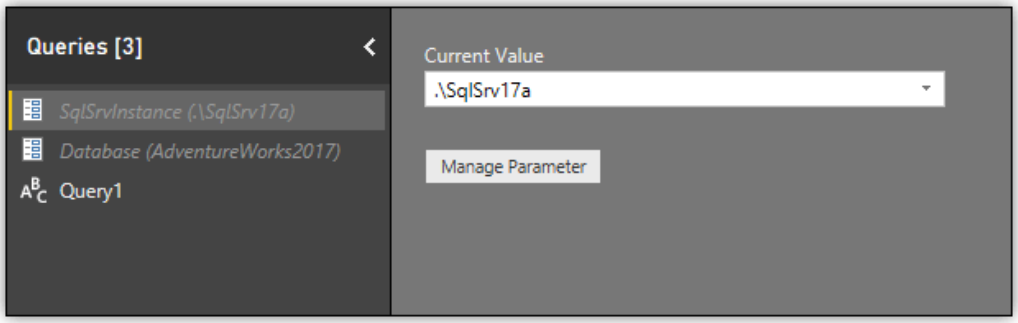
To create the parameter, open Query Editor, click the *Manage Parameters* down arrow on the *Home* ribbon, and then click *New Parameter*. In the *Parameters* dialog box, type *SqlSrvInstance* in the *Name* text box, and then type a parameter description in the *Description* text box.

From the *Type* drop-down list, select *Text,* and from the *Suggested Values* drop-down list, select *List of values*. When you select the *List of values* option, a grid appears, where you type in the individual values you want to assign to the variable, as shown in the following figure. Be sure that at least one of those values is the name of an actual SQL Server instance.



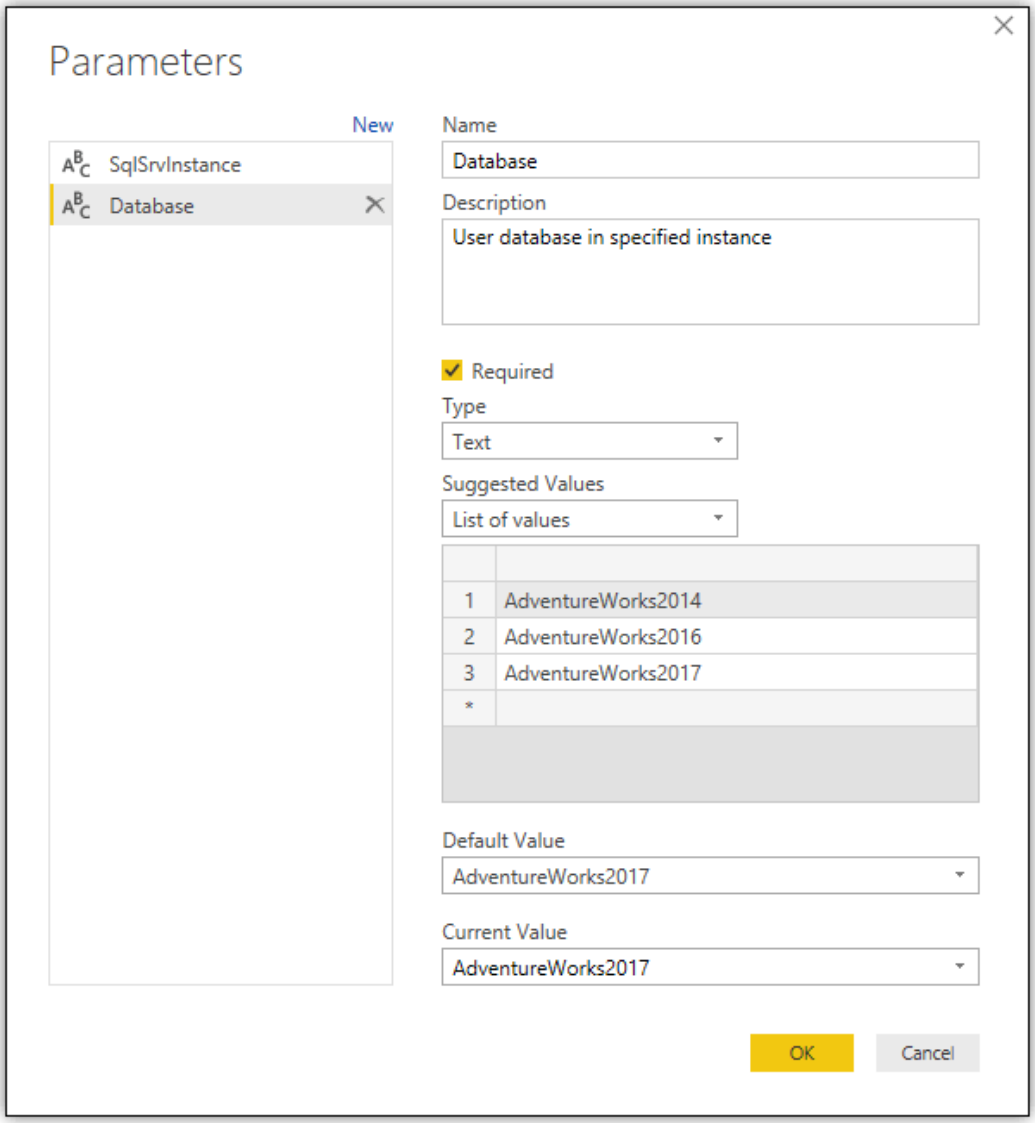
After you type the list of values, select a default value from the *Default Value* drop-down list and then select the variable's current value from the *Current Value* drop-down list. In the figure above, a local SQL Server instance named *SqlSrv17a* is used for both the default and current values.

Click *OK* to close the *Parameters* dialog box. Query Editor adds the parameter to the *Queries* pane, with the current value shown in parentheses. When you select the parameter, Query Editor displays the *Current Value* drop-down list and the *Manage Parameter* button in the main pane, as shown in the following figure.



You can change the current value at any time by selecting a different one from the *Current Value* drop-down list, or you can change the parameter's settings by clicking the *Manage Parameter* button, which returns you to the *Parameters* dialog box.

To create the parameter for the target database, repeat the process, but name the parameter *Database,* as shown in the following figure. Be sure to provide at least one valid database in your list of databases. When you click *OK,* Query Editor adds the parameter to the *Queries* pane, just like it did for the *SqlSrvInstance* parameter.



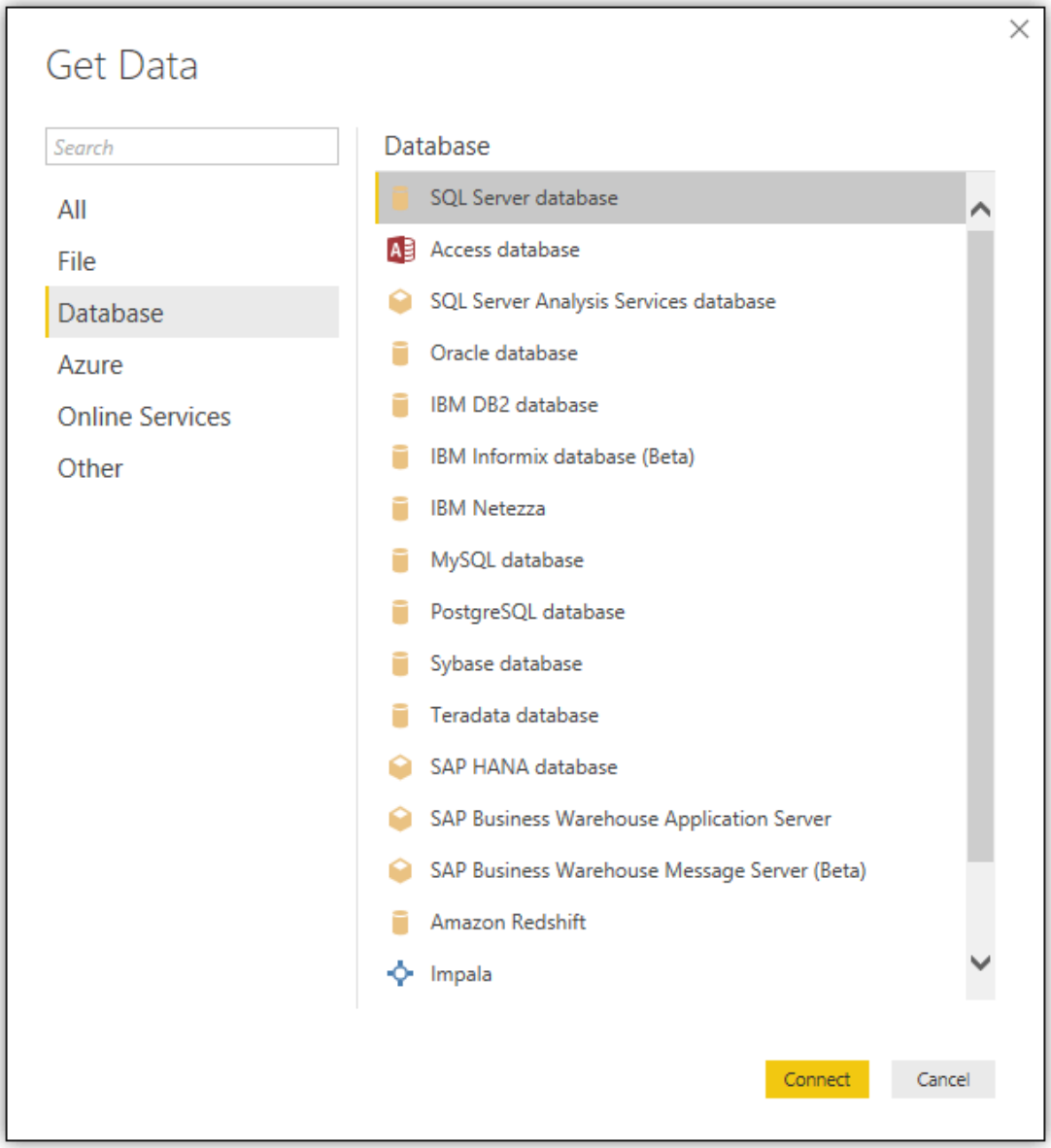
That's all there is to creating the connection parameters. You can also configure parameters with different data types, such as decimals or dates, or with different value formats. For example, you can configure the parameter to accept any value or to use values from a *list query,* a type of query that contains only one column. You can create a list query manually by using an M statement, or you can create a list query based on a regular dataset, although this approach still seems somewhat buggy. For this article, you'll stick with lists, but know you have other options.

Using Parameters to Connect to a Data Source

With your parameters defined, you're ready to connect to the SQL Server instance and retrieve data from the target database. If you haven't already done so, apply your changes in Query Editor and close the window.

For this exercise, you will be running a T-SQL query, but before you try to do this, verify that the *Require user approval for new native database queries* property is disabled. If it is enabled, you will receive an error when trying to run the query. To access the property, click the *File* menu, point to *Options and settings,* and then click *Options*. In the *Options* dialog box, go to the *Security* category, clear the property's checkbox if selected, and click *OK*.

Then, in the main Power BI Desktop window, go to *Data* view and click *Get Data* on the *Home* ribbon. In the *Get Data* dialog box, navigate to the *Database* connections and select *SQL Server database,* as shown in the following figure.

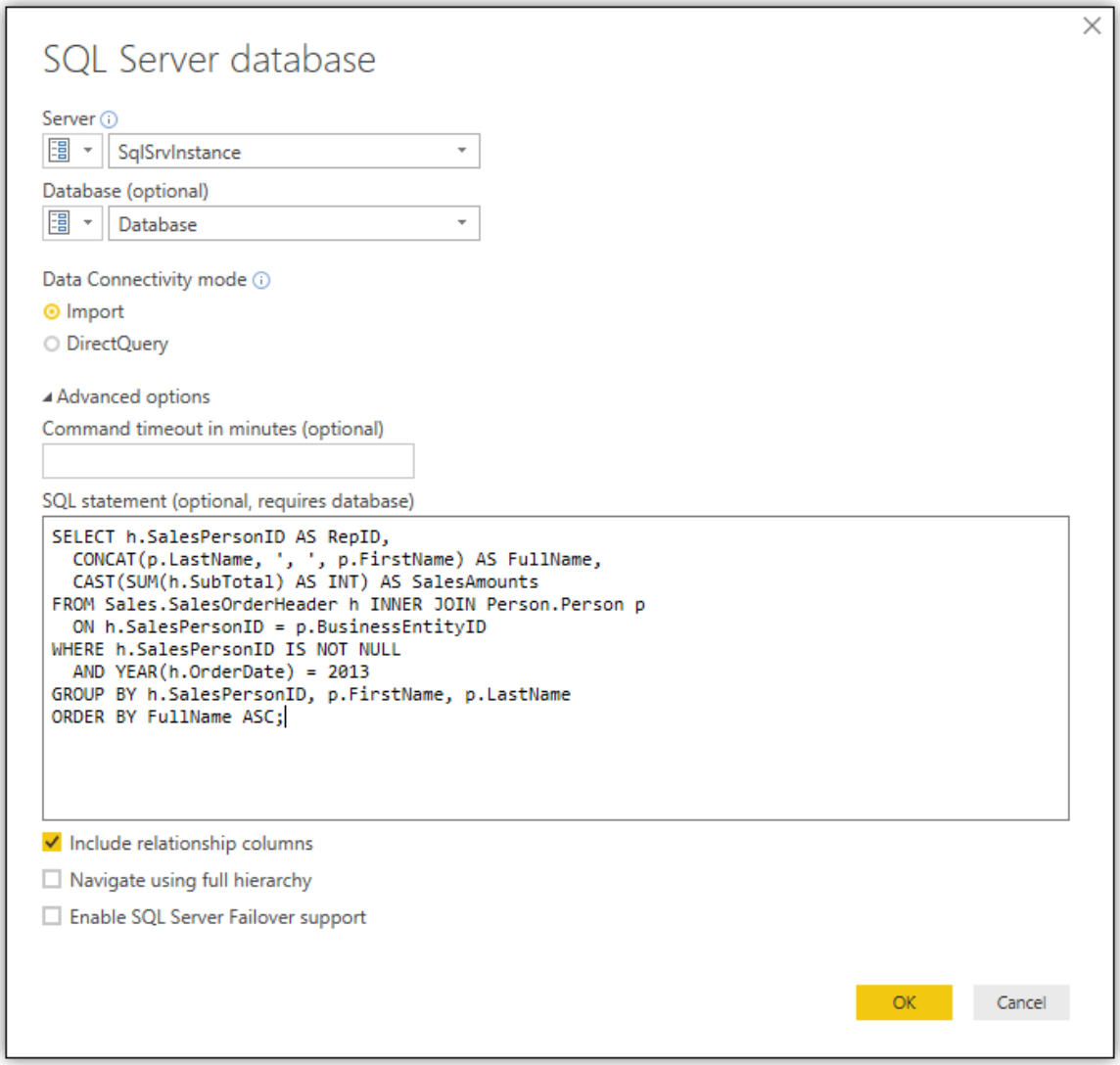


When you click the *Connect* button, the *SQL Server database* dialog box appears. In the *Server* section at the top of the dialog box, click the down arrow associated with the first option (the one to the left) and then click *Parameter*. The second option will change from a text box to a drop-down list that contains the two parameters you just created. Select *SqlSrvInstance,* and then repeat the process in the *Database* section, only this time, select *Database*.

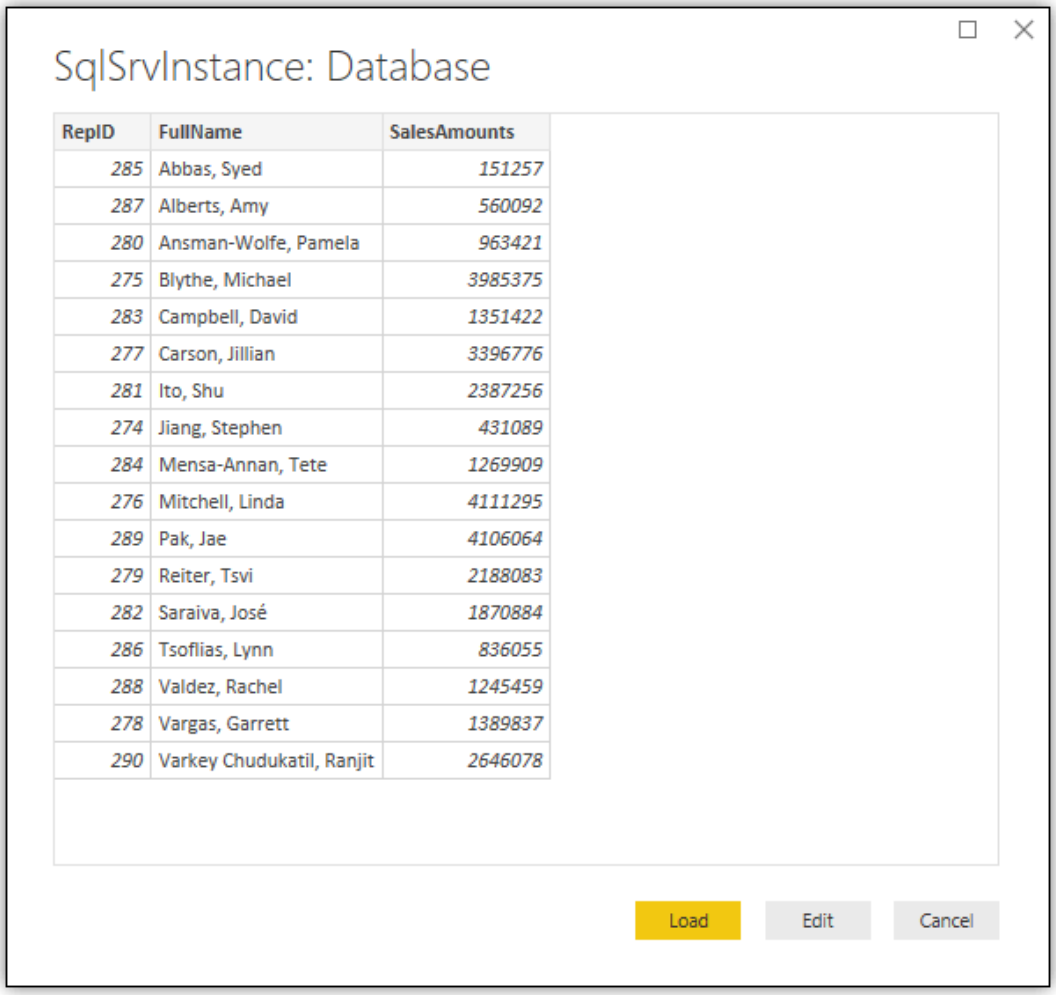
Next, click the *Advanced options* arrow and enter the following T-SQL statement in the *SQL statement* box (or a comparable statement appropriate for your data):

```
SELECT h.SalesPersonID AS RepID,
 CONCAT(p.LastName, ', ', p.FirstName) AS FullName,
 CAST(SUM(h.SubTotal) AS INT) AS SalesAmounts
FROM Sales.SalesOrderHeader h INNER JOIN Person.Person p
 ON h.SalesPersonID = p.BusinessEntityID
WHERE h.SalesPersonID IS NOT NULL
 AND YEAR(h.OrderDate) = 2013
GROUP BY h.SalesPersonID, p.FirstName, p.LastName
ORDER BY FullName ASC;
```

The *SQL Server database* dialog box should now look similar to the one shown in the following figure, with the *SqlSrvInstance* and *Database* parameters selected.

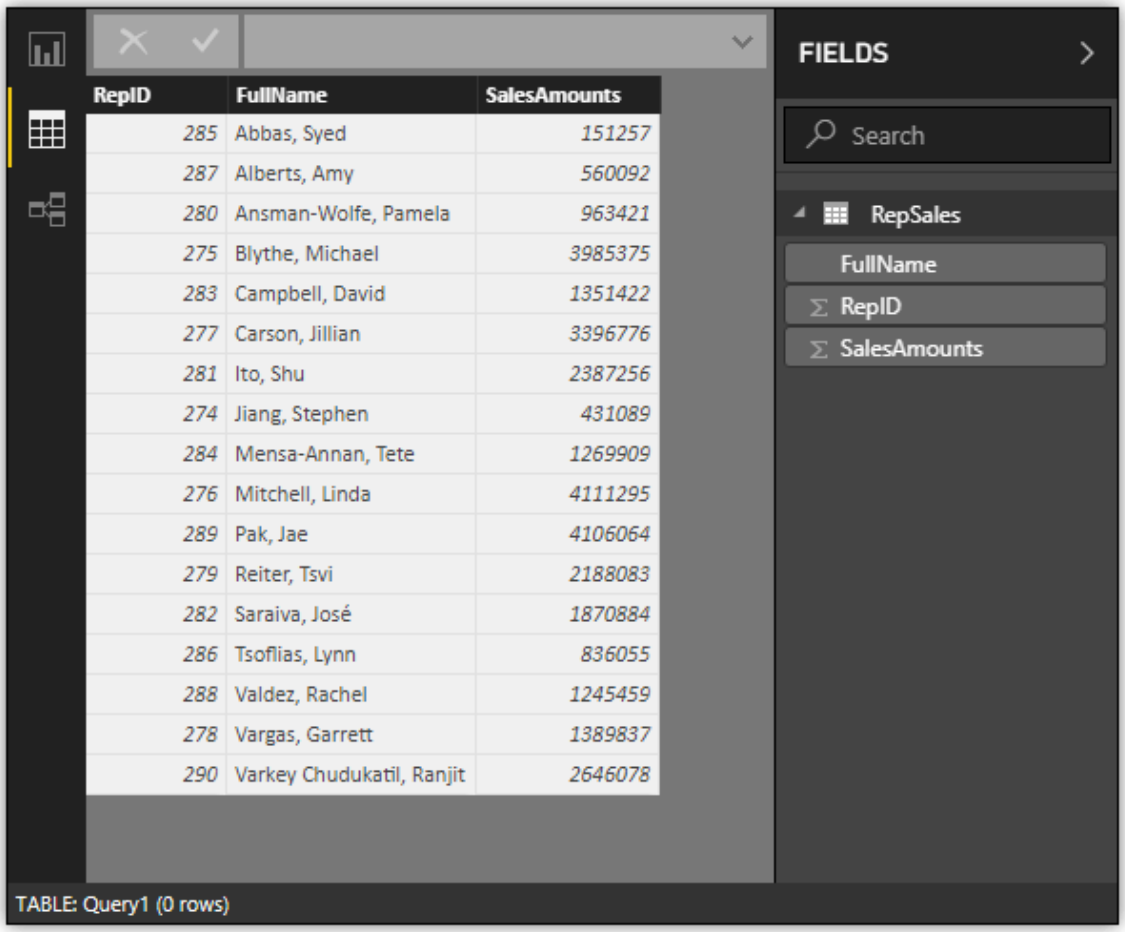


When you click *OK,* Power BI Desktop displays a preview window (assuming everything is working as it should), with the name of the two parameters at the top, as shown in the following figure.



RepID	FullName	SalesAmounts
285	Abbas, Syed	151257
287	Alberts, Amy	560092
280	Ansman-Wolfe, Pamela	963421
275	Blythe, Michael	3985375
283	Campbell, David	1351422
277	Carson, Jillian	3396776
281	Ito, Shu	2387256
274	Jiang, Stephen	431089
284	Mensa-Annan, Tete	1269909
276	Mitchell, Linda	4111295
289	Pak, Jae	4106064
279	Reiter, Tsvi	2188083
282	Saraiva, José	1870884
286	Tsoflias, Lynn	836055
288	Valdez, Rachel	1245459
278	Vargas, Garrett	1389837
290	Varkey Chudukatil, Ranjit	2646078

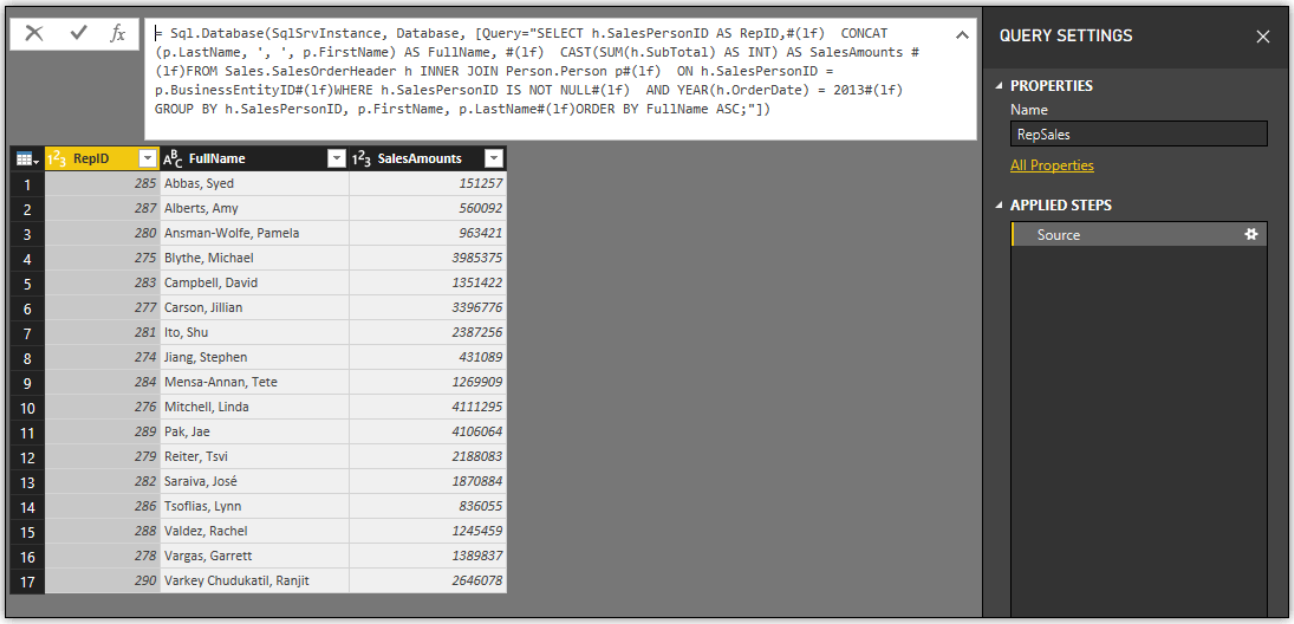
When you click the *Load* button, Power BI Desktop adds the dataset to *Data* view. Before taking any other steps, rename the dataset to *RepSales* or something to your liking. You should end up with a dataset that looks similar to the one shown in the following figure.



RepID	FullName	SalesAmounts
285	Abbas, Syed	151257
287	Alberts, Amy	560092
280	Ansman-Wolfe, Pamela	963421
275	Blythe, Michael	3985375
283	Campbell, David	1351422
277	Carson, Jillian	3396776
281	Ito, Shu	2387256
274	Jiang, Stephen	431089
284	Mensa-Annan, Tete	1269909
276	Mitchell, Linda	4111295
289	Pak, Jae	4106064
279	Reiter, Tsvi	2188083
282	Saraiva, José	1870884
286	Tsoflias, Lynn	836055
288	Valdez, Rachel	1245459
278	Vargas, Garrett	1389837
290	Varkey Chudukatil, Ranjit	2646078

By defining parameters for the connection properties, you can easily change their values at any time, should you need to retrieve the data from a different SQL Server instance or database. You can also use the same parameters for multiple datasets, saving you the trouble of having to repeat connection information each time you create a dataset based on the same data source.

Later in the article, you'll learn more about working with existing parameters after they've been incorporated into your dataset. But for now, open Query Editor and view the M statement associated with the dataset's *Source* step in the *Applied Steps* section, as shown in the following figure.



	RepID	FullName	SalesAmounts
1	285	Abbas, Syed	151257
2	287	Alberts, Amy	560092
3	280	Ansman-Wolfe, Pamela	963421
4	275	Blythe, Michael	3985375
5	283	Campbell, David	1351422
6	277	Carson, Jillian	3396776
7	281	Ito, Shu	2387256
8	274	Jiang, Stephen	431089
9	284	Mensa-Annan, Tete	1269909
10	276	Mitchell, Linda	4111295
11	289	Pak, Jae	4106064
12	279	Reiter, Tsvi	2188083
13	282	Saraiva, José	1870884
14	286	Tsoflias, Lynn	836055
15	288	Valdez, Rachel	1245459
16	278	Vargas, Garrett	1389837
17	290	Varkey Chudukatil, Ranjit	2646078

If you examine the M statement closely, you'll see that the SQL Server connection data references the *SqlSrvInstance* and *Database* parameters. They're included as the first two arguments to the **Sql.Database** function. I've copied the statement here and highlighted the two parameters for easier viewing:

```
= Sql.Database(SqlSrvInstance, Database, [Query="SELECT h.SalesPersonID AS RepID,#(lf)  CONCAT(p.LastName, ', ', p.FirstName) AS FullName, #(lf)  CAST(SUM(h.SubTotal) AS INT) AS SalesAmounts #(lf)FROM Sales.SalesOrderHeader h INNER JOIN Person.Person p#(lf)  ON h.SalesPersonID = p.BusinessEntityID#(lf)WHERE h.SalesPersonID IS NOT NULL#(lf)  AND YEAR(h.OrderDate) = 2013#(lf)GROUP BY h.SalesPersonID, p.FirstName, p.LastName#(lf)ORDER BY FullName ASC;"])
```

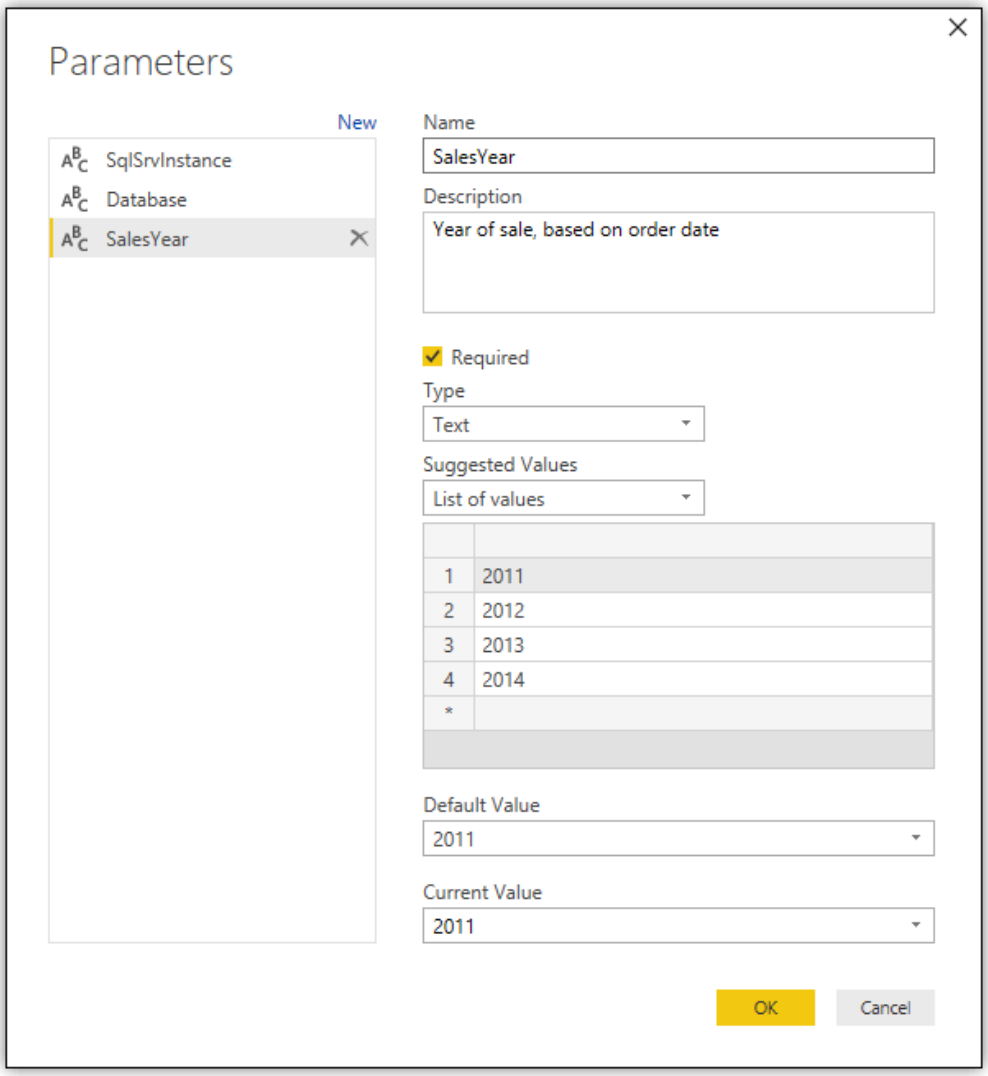
An interesting implication of all this is how easily you can reference parameters within your M statements, providing you with a very powerful and flexible tool for customizing the applied steps that make up your dataset.

Adding Parameters to Filter Data

In some cases, you might want to use parameters to filter a dataset, rather than applying a *Filtered Rows* step, an approach that can be inflexible and difficult to update. For example, you might want to use a parameter in place of the hard-coded **2013** specified the **WHERE** clause of the original T-SQL statement:

```
YEAR(h.OrderDate) = 2013
```

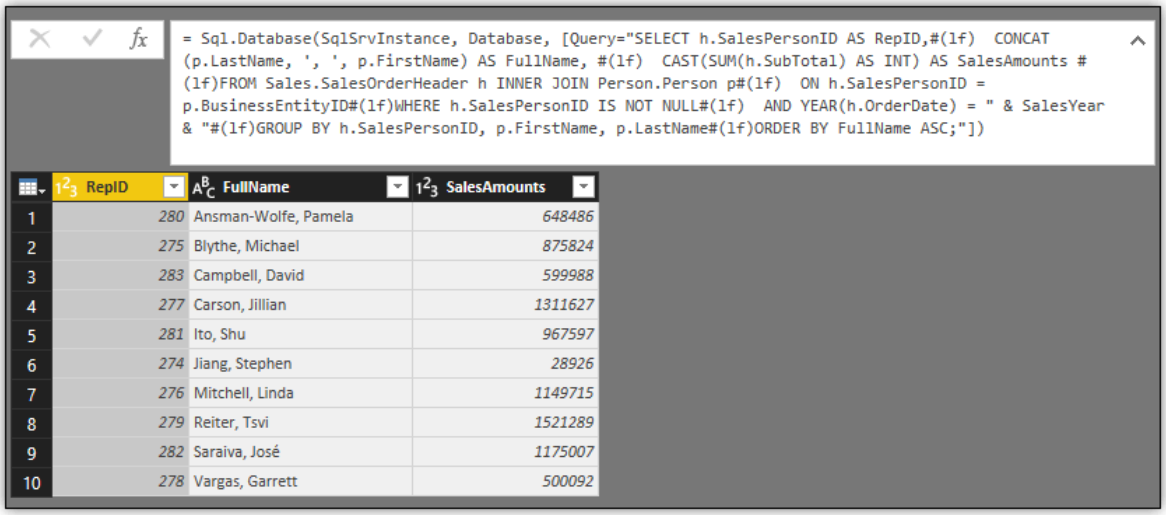
You can replace the hard-coded value with a parameter that supports a range of values. To do so, create a parameter just like you saw earlier, only this time, name the parameter *SalesYear,* define a list that contains the years *2011* through *2014,* and set the default and current values to *2011,* as shown in the following figure.



After you've created the parameter, update the M statement associated with the dataset's *Source* step by replacing the **2013** value with the following code snippet (including the quotation marks):

```
" & SalesYear & "
```

Once you've updated the code, click the checkmark to the left of the statement to verify that your changes are formatted correctly and that you didn't somehow break the code. The statement and dataset should now look similar to those shown in the following figure.



	RepID	FullName	SalesAmounts
1	280	Ansman-Wolfe, Pamela	648486
2	275	Blythe, Michael	875824
3	283	Campbell, David	599988
4	277	Carson, Jillian	1311627
5	281	Ito, Shu	967597
6	274	Jiang, Stephen	28926
7	276	Mitchell, Linda	1149715
8	279	Reiter, Tsvi	1521289
9	282	Saraiva, José	1175007
10	278	Vargas, Garrett	500092

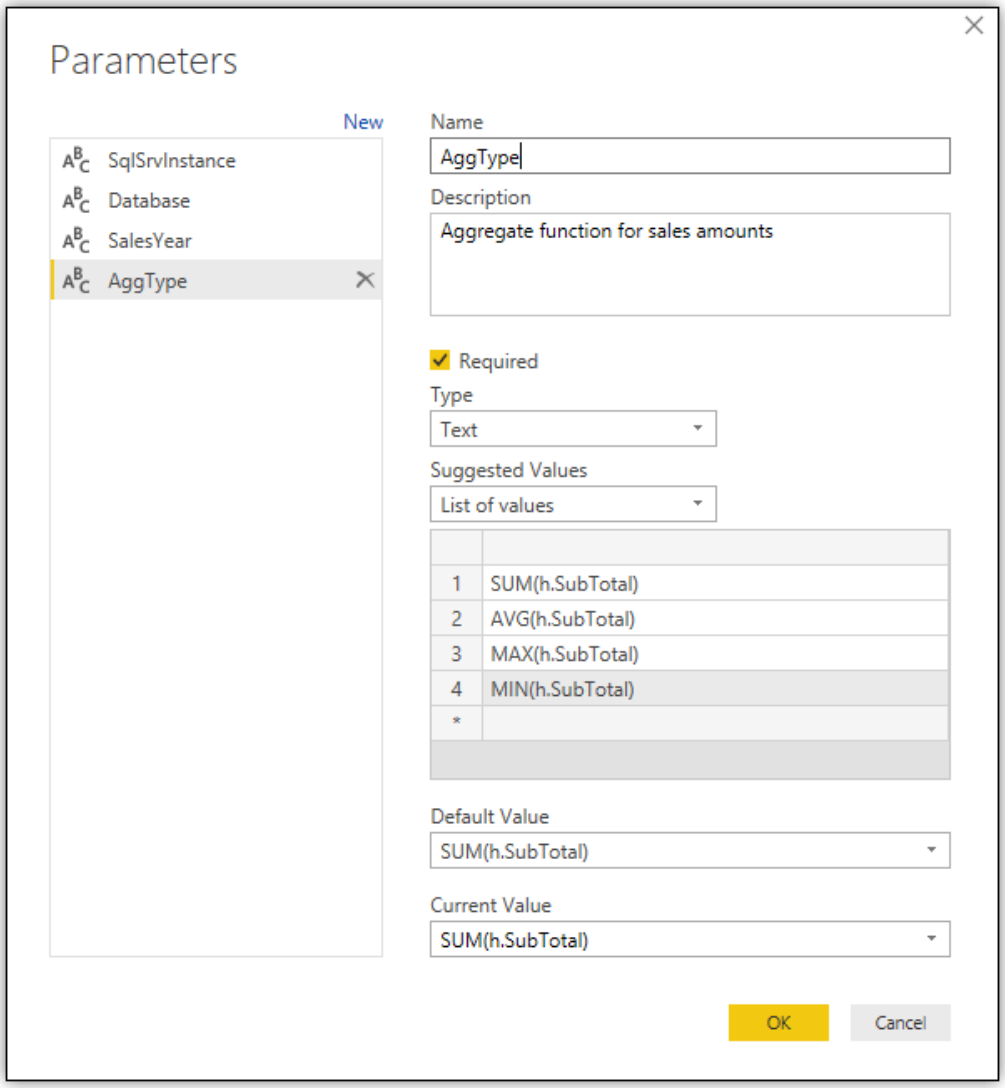
To test the new variable, select *SalesYear* in the *Queries* pane and choose a different year from the default *2011*. Then re-select the *RepSales* dataset and verify that the data has been updated.

Adding Parameters to Control Statement Logic

In some cases, you might want to use parameters to control a query's logic, rather than just filtering data. For example, the **SELECT** clause in the original T-SQL statement uses the **SUM** aggregate function to come up with the total sales for each sales rep:

```
CAST(SUM(h.SubTotal) AS INT) AS SalesAmounts
```

You can instead insert a parameter in the M statement that allows you to apply a different aggregate function. First, create a parameter name *AggType* and then define a list that includes one item for each function, using the **SUM** function as the default and current values, as shown in the following figure.

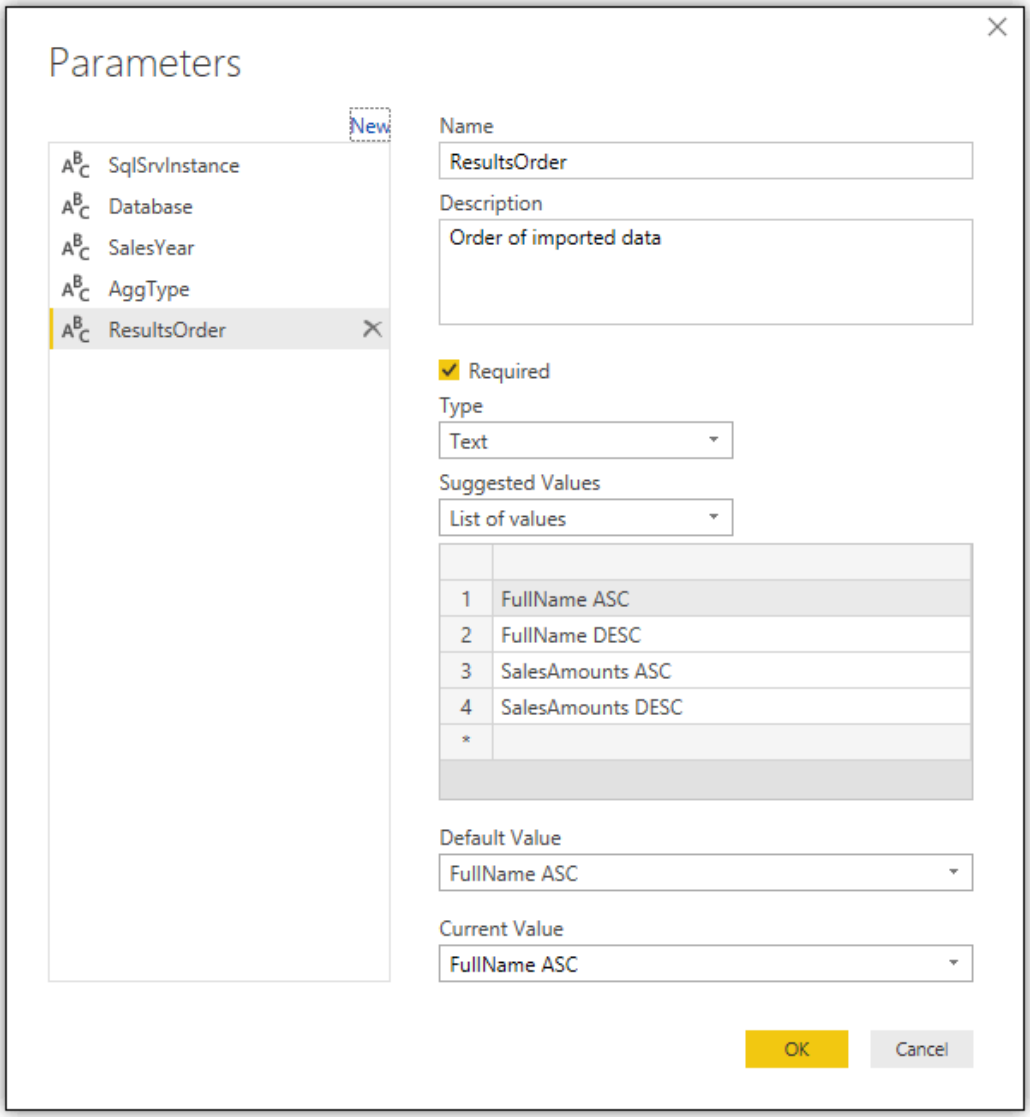


I've included the **SubTotal** column within the options in part to keep the logic clearer, but also to demonstrate that you can summarize the data based on other columns as well, as long as your dataset supports them. For example, you dataset might also include the **DiscountAmounts** column, which provides the total sales amounts, less any discounts. In such cases, each parameter option can define the specific function and column, including such values as **SUM(h.DiscountAmounts)** and **AVG(h.SubTotal)**.

After you create the parameter, update the M statement associated with the dataset's *Source* step by replacing the hard-coded **SUM(h.SubTotal)** fragment with the following snippet (including the quotes):

```
" & AggType & "
```

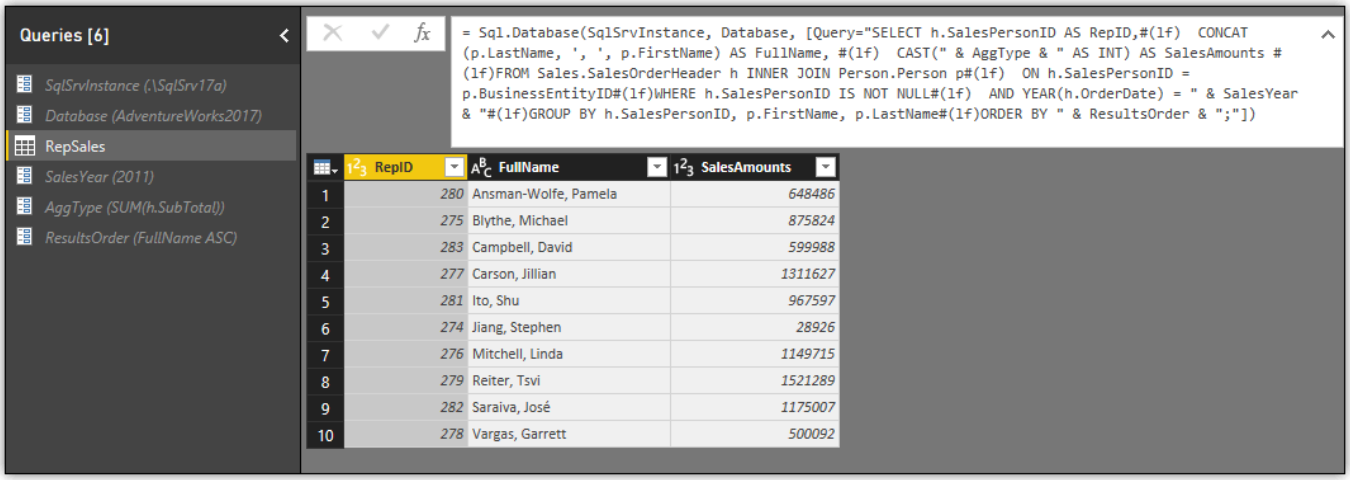
You can also do something similar with the original **ORDER BY** clause, using a variable to provide options for how to order the data. First, create a variable named *ResultsOrder.* Then, for each option in the list, specify the column on which to base the sorting (*FullName* or *SalesAmounts*) and whether the sorting should be done in ascending or descending order, as shown in the following figure. In this case, the option *FullName ASC* is used for the default and current values.



Once you've created the parameter, update the M statement by replacing the **FullName ASC** code fragment with the following snippet (including the quotes):

```
" & ResultsOrder & "
```

Your M statement and dataset should now look similar to the ones shown in the following figure.



Once you understand how to incorporate parameters into your M statements, you have a wide range of options for manipulating data and making your datasets more flexible and accommodating, without having to add a lot of steps to you query. Just be sure that whenever you make any changes, you apply and save them so you know they work and won't get lost.

Using Parameters to Name Dataset Object

Another fun thing you can do with parameters is to use them for applying dynamic names to objects. For example, you can change the name of the *SalesAmounts* column to one that reflects the sales year and type of aggregation being applied. A simple way to do this is to first add a *Renamed Columns* step to your dataset in Query Editor and then update the associated M statement to include the parameter values.

To add the *Renamed Columns* step, right-click the *SalesAmounts* column header, click *Rename,* type *Sales* as a temporary column name, and then press *Enter*. Query Editor adds the *Renamed Columns* step to the *Applied Steps* section, along with the following M statement:

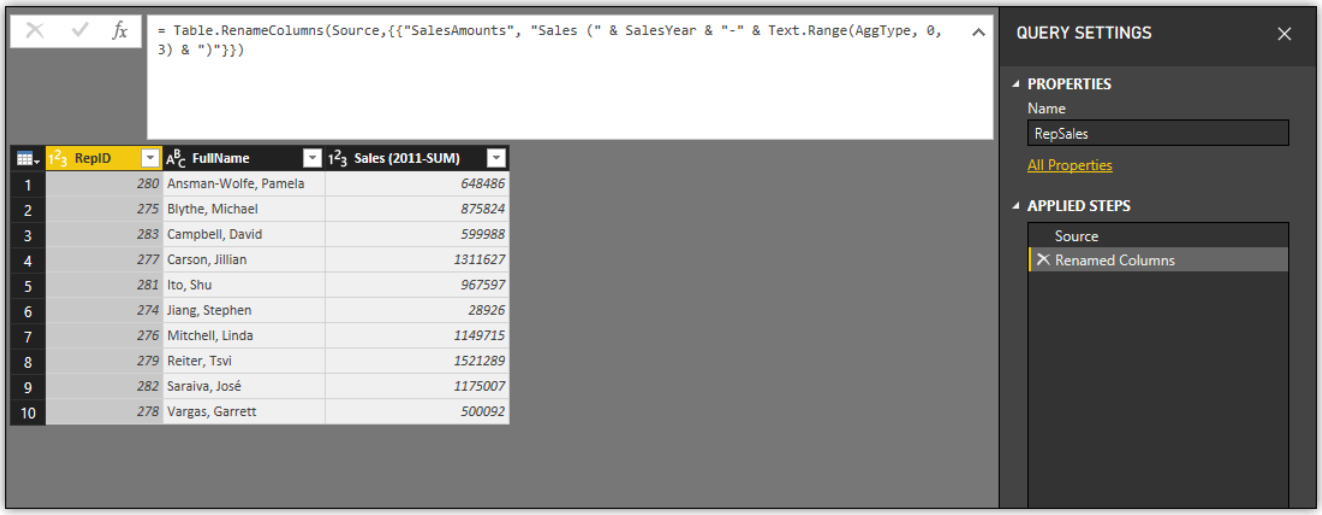
```
= Table.RenameColumns(Source,{{"SalesAmounts", "Sales"}})
```

To incorporate the *SalesYear* and *AggType* parameters into the statement, replace **Sales** with the following code:

```
Sales (" & SalesYear & "-" & Text.Range(AggType, 0, 3) & ")
```

The concatenation operator (&) joins the name *Sales* with the two variable values, which are separated by a dash and enclosed in parentheses. The **Text.Range** function retrieves only the first three characters from the *AggType* variable.

After you update the statement, be sure to verify your changes. The dataset and M statement should now look similar to the ones shown in the following figure.

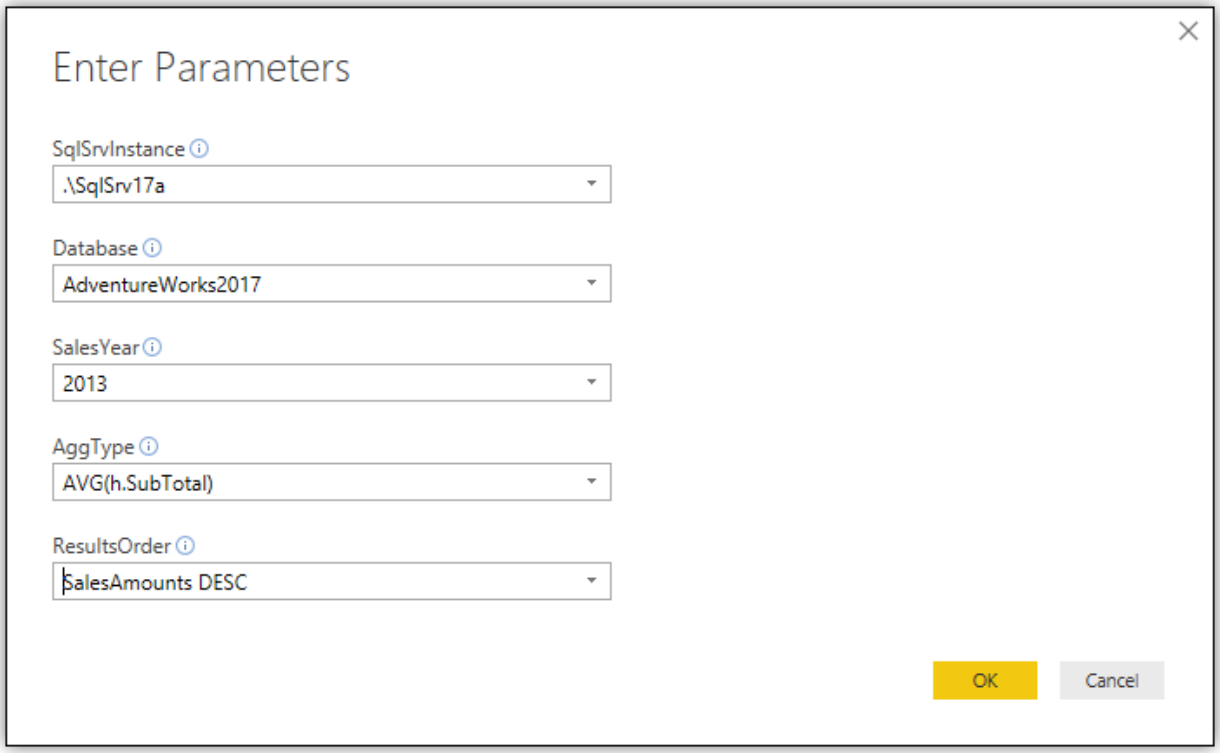


Notice that the name of the dataset's third column now includes the year and function in parentheses, making it easy to see what parameter values have been applied to the data set.

Working with Parameters in Data View

As you saw earlier, to change a parameter value in Query Editor, you need to select the parameter in the *Queries* pane and update its value accordingly. However, this can be a cumbersome process, especially when you want to update multiple parameter values concurrently.

Fortunately, you can set parameter values directly in *Data* view. To set the values, click the *Edit Queries* down arrow on the *Home* ribbon and then click *Edit Parameters*. When the *Enter Parameters* dialog box appears, select the values you want to apply to your datasets and then click *OK.* The following figure shows the *Enter Parameters* dialog box and the current parameter settings.



Although the *Edit Parameters* option name and the *Enter Parameters* dialog box name are somewhat misleading, the features they represent provide an effective way to update the parameter values and in turn update the data. Be aware, however, that the changes you make here apply to all datasets using the parameters. If you want to include similar types of parameters in multiple datasets, but you do not what them to all share the same values, you should create parameters specific to the dataset, naming them in such a way to make them easily distinguishable from each other.

Once you know how to set the parameter values, you can try different variations, viewing the dataset each time you apply the new settings. For example, the following figure shows the dataset after applying the parameter settings shown in the previous figure.

RepID	FullName	Sales (2013-AVG)
280	Ansman-Wolfe, Pamela	50706
274	Jiang, Stephen	30792
290	Varkey Chudukatil, Ranjit	28150
276	Mitchell, Linda	25378
281	Ito, Shu	24360
289	Pak, Jae	24153
275	Blythe, Michael	22774
282	Saraiva, José	21754
287	Alberts, Amy	19314
283	Campbell, David	18770
277	Carson, Jillian	18361
278	Vargas, Garrett	15616
284	Mensa-Annan, Tete	15487
288	Valdez, Rachel	14482
279	Reiter, Tsvi	13762
286	Tsoflias, Lynn	12668
285	Abbas, Syed	12605

Notice that the *Sales* column includes the year and aggregation type in the column name and that the data is sorted based on the *Sales* values, in descending order. You can also change the parameter values while in *Report* view, allowing you to see you changes immediately within your visualizations.

Clearly, the parameter capabilities help to make Power BI Desktop an even more robust and flexible tool, and you can use parameters in a variety of ways, regardless of where the data originates. The more comfortable you become working with parameters, the more effectively you can use them, and the more control you'll have over your datasets.

Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5

Data for Power BI can come from many sources, but often the data is hosted in SQL Server. In this article, the fifth of the series, Robert Sheldon demonstrates how to work with SQL Server data in Power BI Desktop, including relationships between tables.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Power BI Desktop provides a far more robust environment for developing business intelligence reports than the Power BI service. Power BI Desktop supports more data sources and offers more tools for transforming the data retrieved from those sources.

One data source available to Power BI Desktop that is not available to the service is SQL Server. Although the Power BI service provides connectors to Azure SQL Database and SQL Data Warehouse, it does not offer one for SQL Server. With Power BI Desktop, you can retrieve SQL Server data from entire tables or run queries that return a subset of data from multiple tables. You can also call stored procedures, including the **sp_execute_external_script** stored procedure, which allows you to run Python and R scripts.

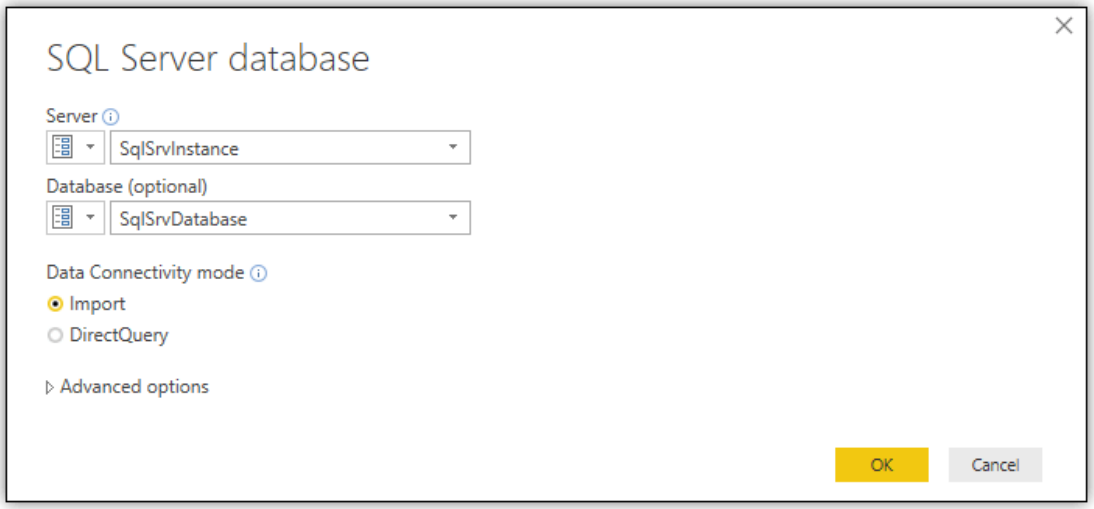
This article walks you through the process of retrieving data from a SQL Server instance, using different methods for returning the data. The article also covers some basic transformations. The examples are based on the **AdventureWorks2017** database, running on a local instance of SQL Server, but the principles apply to any SQL Server database.

In fact, many of the topics covered here can apply to other relational database management systems and even other types of data sources. Because Power BI Desktop organizes all data into *queries* (table-like structures for working with datasets), once you get the data into Power BI Desktop, many of the actions you carry out are similar despite the source. That said, you'll still find features specific to relational databases, particularly when it comes to the relationships between tables.

Retrieving SQL Server Tables

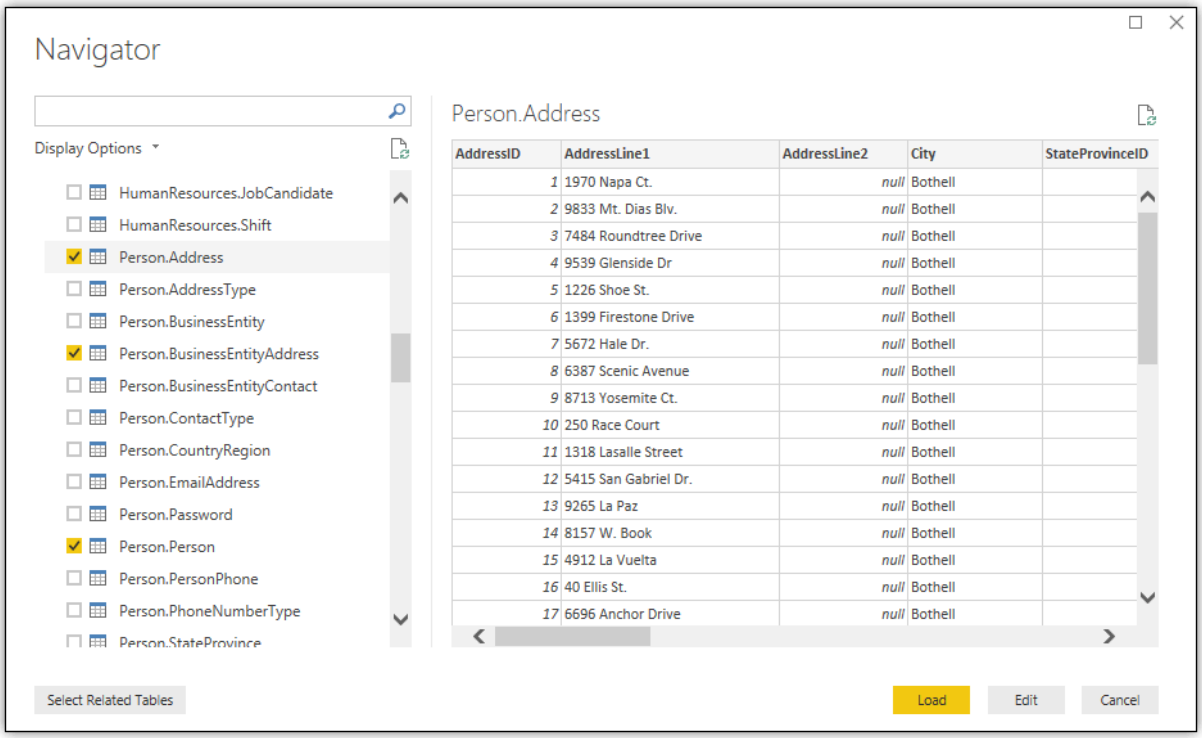
In Power BI Desktop, you can retrieve SQL Server tables or views in their entirety. Power BI Desktop preserves their structures and, where possible, identifies the relationships between them. This section provides an example that demonstrates how importing tables works. If you follow along, you'll start by retrieving three tables from the **AdventureWorks2017** database.

To retrieve the tables, click *Get Data* on the *Home* ribbon in the main Power BI Desktop window. When the *Get Data* dialog box appears, navigate to the *Database* category and double-click *SQL Server database*. In the *SQL Server database* dialog box, provide the name of the SQL Server instance and the target database, as shown in the following figure.



For this article, I created two connection parameters. The first parameter, *SqlSrvInstance,* contains the name of the SQL Server instance. The second parameter, *SqlSrvDatabase,* contains the name of the database. When providing the connection information, you can use the actual names or you can create your own parameters. For details about creating parameters, refer to the previous article in this series.

Once you've entered the necessary information, click *OK*. When the *Navigator* dialog box appears, select the *Person.Address, Person.BusinessEntityAddress,* and *Person.Person* tables, as shown in the following figure.



You can review the data from the selected table in the right pane. You can also choose to include related tables by clicking the *Select Related Tables* button. For example, if you select the *Person* table and then select the related tables, you would end up with nine tables, a lot more than you need for this example. Whether you take advantage of this option will depend on your specific project. If you do, but don't want to keep all the selected tables, simply clear the checkboxes associated with those you want to exclude.

When you're ready to import the tables into Power BI Desktop, click the *Load* button. Any tables and views you've selected will be loaded into Power BI Desktop and listed as individual datasets in *Data* view. From there, you can modify the datasets in Query Editor, just as you would data retrieved from any other source.

One of the first steps you're likely to take in Query Editor is to rename each dataset and from there, remove any extra columns:

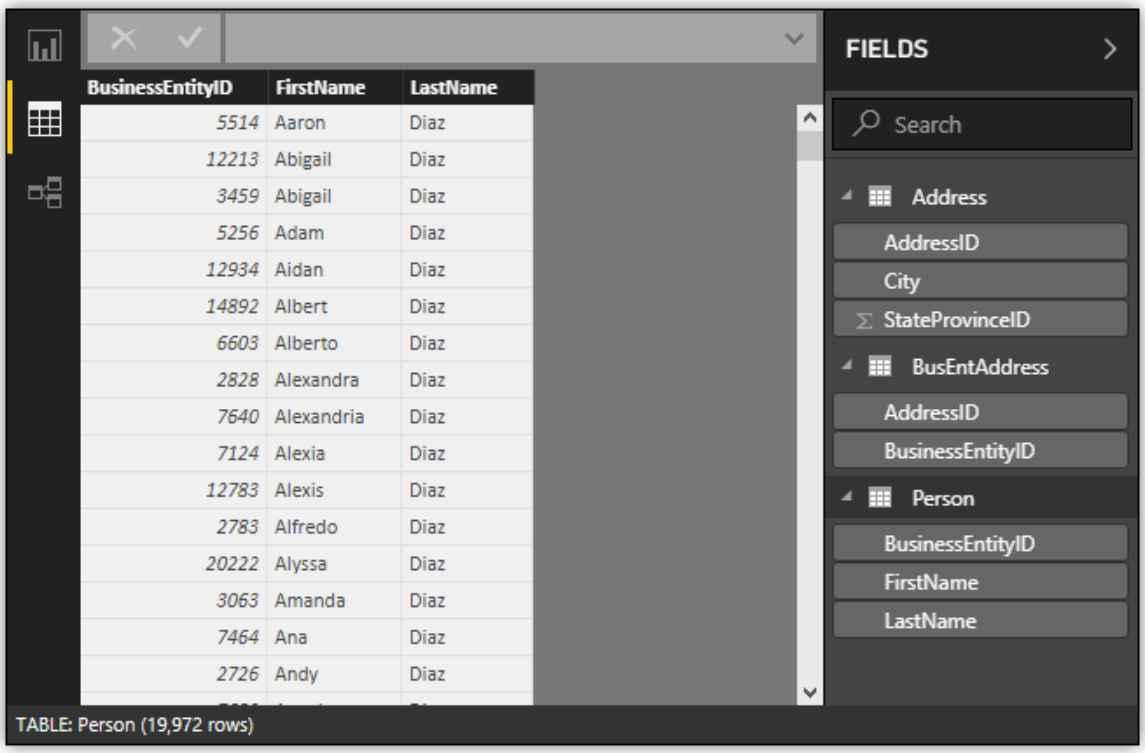
- To rename a dataset, right-click the dataset in the *Queries* pane, click *Rename,* type in the new name, and press *Enter*.
- To remove a column, select the column in the main grid and click *Remove Columns* on the *Home* ribbon.
- To remove multiple columns in one operation, select the first column, press and hold *Control* as you select each additional column, and then click *Remove Columns* on the *Home* ribbon.

With these instruction in mind, take the following steps for the three tables you just imported:

- Change the name of the *Person Address* table to *Address,* and then remove the columns *AddressLine1, AddressLine2, PostalCode, SpatialLocation, rowguid,* and *ModifiedDate*.
- Change the name of the *Person BusinessEntityAddress* table to *BusEntAddress,* and then remove the columns *AddressTypeID, rowguid,* and *ModifiedDate*.
- Change the name of the *Person Person* table to *Person,* and then remove the columns *PersonType, NameStyle, Title, MiddleName, Suffix, EmailPromotion, AdditionalContactInfo, Demographics, rowguid,* and *ModifiedDate*.

For now, ignore the columns that contain relationship data (the columns with the gold-colored values). We'll get back to those shortly.

When you're finished updating the queries, apply your changes and close Query Editor. Your datasets should look like those shown in the following figure, as they appear in *Data* view. In this case, the *Person* dataset is selected.



Notice that each dataset now includes only two or three fields. You can also add filters in Query Editor to refine the data even more, or you can add filters to individual visualizations when you're building your reports.

Working with Relationships

When you updated your datasets in Query Editor, you no doubt noticed that the queries included a number of *pseudo-columns,* indicated by the long column names and gold-colored values, which are either *Table* or *Value*.

The columns reflect the relationships that exist between the between tables in the source database. A column that contains *Table* values represents a foreign key in another table that references the primary table on which the query is based. A column that contains *Value* values represents a foreign key in the primary table that references another table.

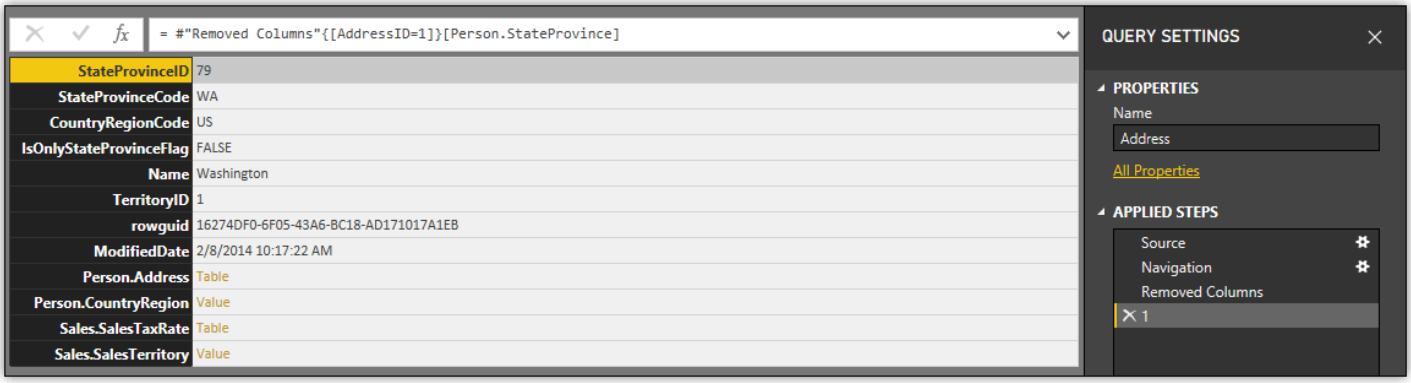
For example, when you select the *Address* query in Query Editor, you should see four relationship columns, as shown in the following figure.

	AddressID	City	StateProvinceID	Person.BusinessEntityAddress	Person.StateProvince	Sales.SalesOrderHeader(AddressI...	Sales.SalesOrderHeader(AddressID) 2
1	1	Bothell	79	Table	Value	Table	Table
2	2	Bothell	79	Table	Value	Table	Table
3	3	Bothell	79	Table	Value	Table	Table
4	4	Bothell	79	Table	Value	Table	Table
5	5	Bothell	79	Table	Value	Table	Table
6	6	Bothell	79	Table	Value	Table	Table
7	7	Bothell	79	Table	Value	Table	Table
8	8	Bothell	79	Table	Value	Table	Table
9	9	Bothell	79	Table	Value	Table	Table
10	10	Bothell	79	Table	Value	Table	Table
11	11	Bothell	79	Table	Value	Table	Table
12	12	Bothell	79	Table	Value	Table	Table
13	13	Bothell	79	Table	Value	Table	Table
14	14	Bothell	79	Table	Value	Table	Table
15	15	Bothell	79	Table	Value	Table	Table
16	16	Bothell	79	Table	Value	Table	Table

The columns indicate that the **Address** table, as it exists in the **AdventureWorks2017** database, contains one foreign key and is referenced by three foreign keys:

- The *Person.BusinessEntityAddress* column in the *Address* query represents a foreign key in the **Person.BusinessEntityAddress** table that references the **Person.Address** table. The key is created on the **AddressID** column in the **Person.BusinessEntityAddress** table.
- The *Person.StateProvince* column in the *Address* query represents a foreign key in the **Person.Address** table that references the **Person.StateProvince** table. The key is created on the **StateProvinceID** column in the **Person.Address** table.
- The *Sales.SalesOrderHeader(AddressID)* column in the *Address* query represents a foreign key in the **Sales.SalesOrderHeader** table that references the **Person.Address** table. This key is created on the **BillToAddress** column in the **Sales.SalesOrderHeader** table.
- The *Sales.SalesOrderHeader(AddressID) 2* column in the *Address* query represents a foreign key in the **Sales.SalesOrderHeader** table that references the **Person.Address** table. This key is created on the **ShipToAddress** column in the **Sales.SalesOrderHeader** table.

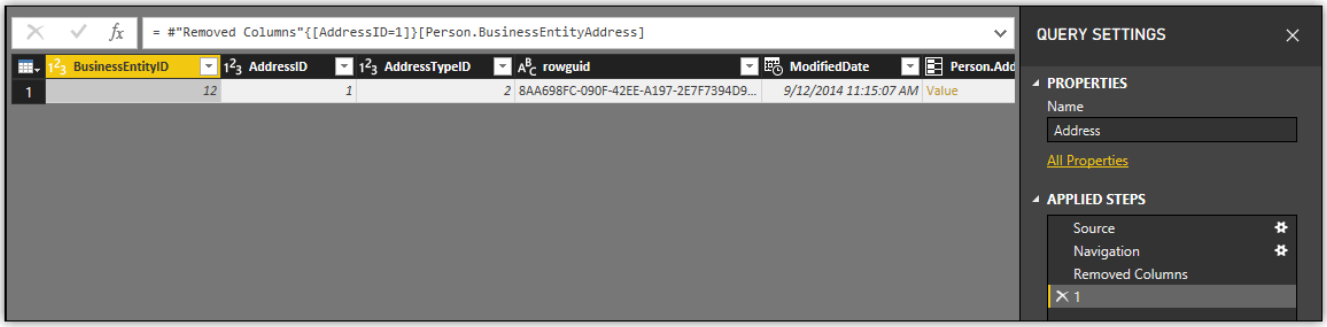
If you click a *Value* or *Table* value, Query Editor will add a step to the *Applied Steps* section in the right pane and modify the dataset to include information specific to the selected value. For example, the first row in the *Address* query contains an *AddressID* value of 1 and a *StateProvinceID* value of 79. If you click the *Value* value in the *Person.StateProvince* column for that row, Query Editor will add a step named *1* to *Applied Steps* and modify the dataset to include only information from the *StateProvince* table specific to the *StateProvinceID* value of *79,* as shown in the following figure.



You can click any instance of *Value* in the *Person.StateProvince* column to see the related data. The associated step in the *Applied Steps* section will be named after the *AddressID* value of the selected row. When you are finished viewing the results, you can return the query to its previous state by deleting the added steps in the *Applied Steps* section.

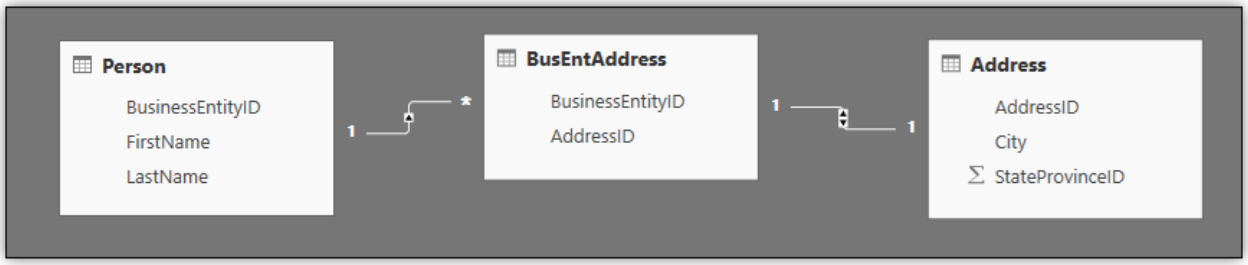
If you were to now click the *Table* value in the *Person.BusinessEntityAddress* column for the first row, Query Editor would again add a step named *1*, only this time the information would come from the *BusinessEntityAddress* table, and the modified dataset

would include only one row, as shown in the following figure. The row includes a foreign key reference to the **AddressID** value of **1** in the **Address** table.

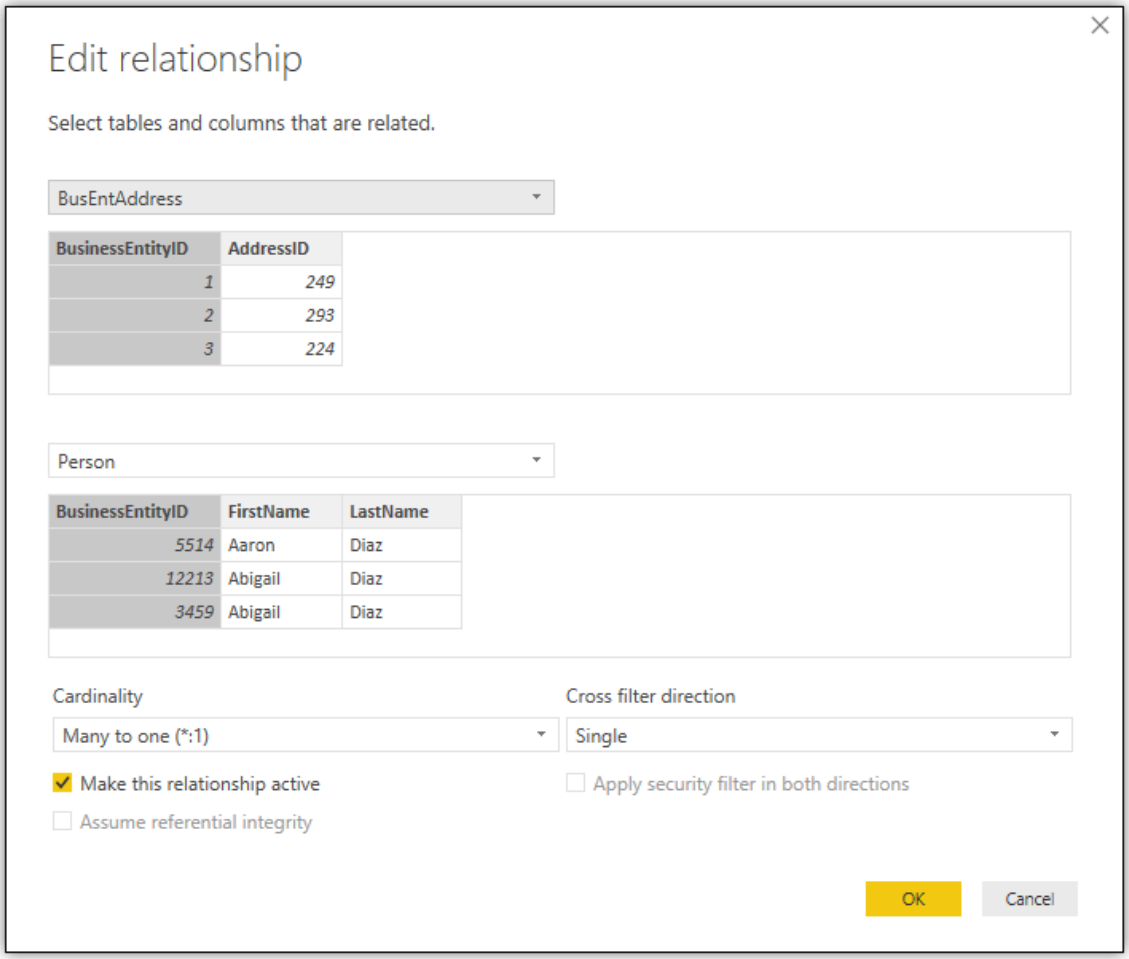


You can play with the *Value* and *Table* values as much as you like to see the different related data. Just be sure to remove the step from *Applied Steps* to return the query to its previous state. Although you won't be using this feature for the examples in this article, you might find it useful for other projects based on SQL Server data.

In addition to identifying foreign key relationships, Power BI Desktop also attempts to identify the relationships between the imported datasets. You can view these relationships by going to the *Relationships* pane in the main Power BI Desktop window. There you'll find a graphically representation of each dataset, which you can move around as necessary. If Power BI Desktop detects a relationship between two datasets, they're joined by a connector that indicates the type of relationship, as shown in the following figure.

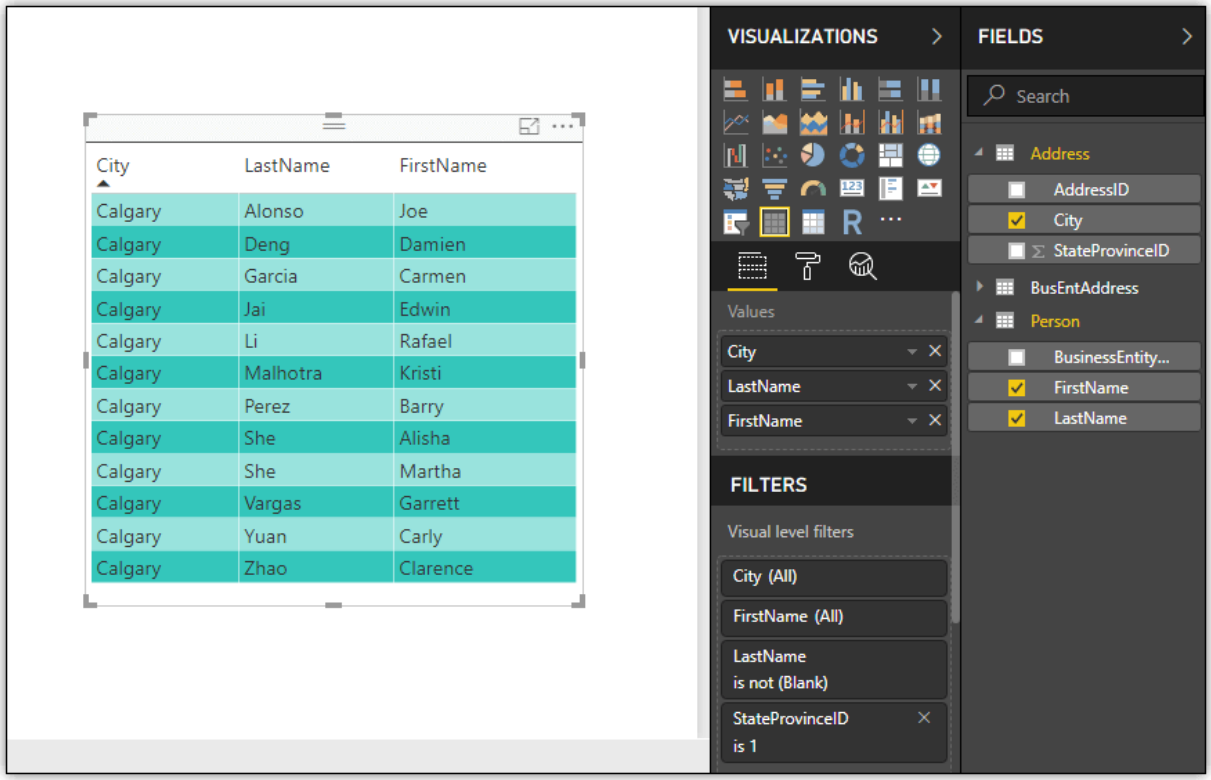


In this case, Power BI Desktop has detected a one-to-many relationship between the *Person* and *BusEntAddress* datasets and a one-to-one relationship between the *BusEntAddress* and *Address* datasets. You can view details about a relationship by double-clicking the connector. For example, to view details about the relationship between the *Person* and *BusEntAddress* datasets, double-click the connector that joins the datasets. This launches the *Edit relationship* dialog box, shown in the following figure.



In the dialog box, you can view the details about the relationship and even modify them if necessary, although Power BI Desktop does a pretty good job at getting the relationships right, based on the current data in the dataset. Notice that the dialog box shows this as a many-to-one relationship, rather than one-to-many. This is because the *BusEntAddress* dataset is listed first. You can find details about defining and editing relationships, as well as the meaning of the individual options, in the Microsoft article [Create and manage relationships in Power BI Desktop](#).

The fact that Power BI Desktop detects relationships makes it easier to create visualizations based on multiple datasets. For example, I created a simple table visualization in *Report* view based on data from the *Person* and *Address* datasets, without having to map data across all three datasets. The following figure shows the table with a filter applied on the visualization (the *StateProvinceID* value set to *1*).

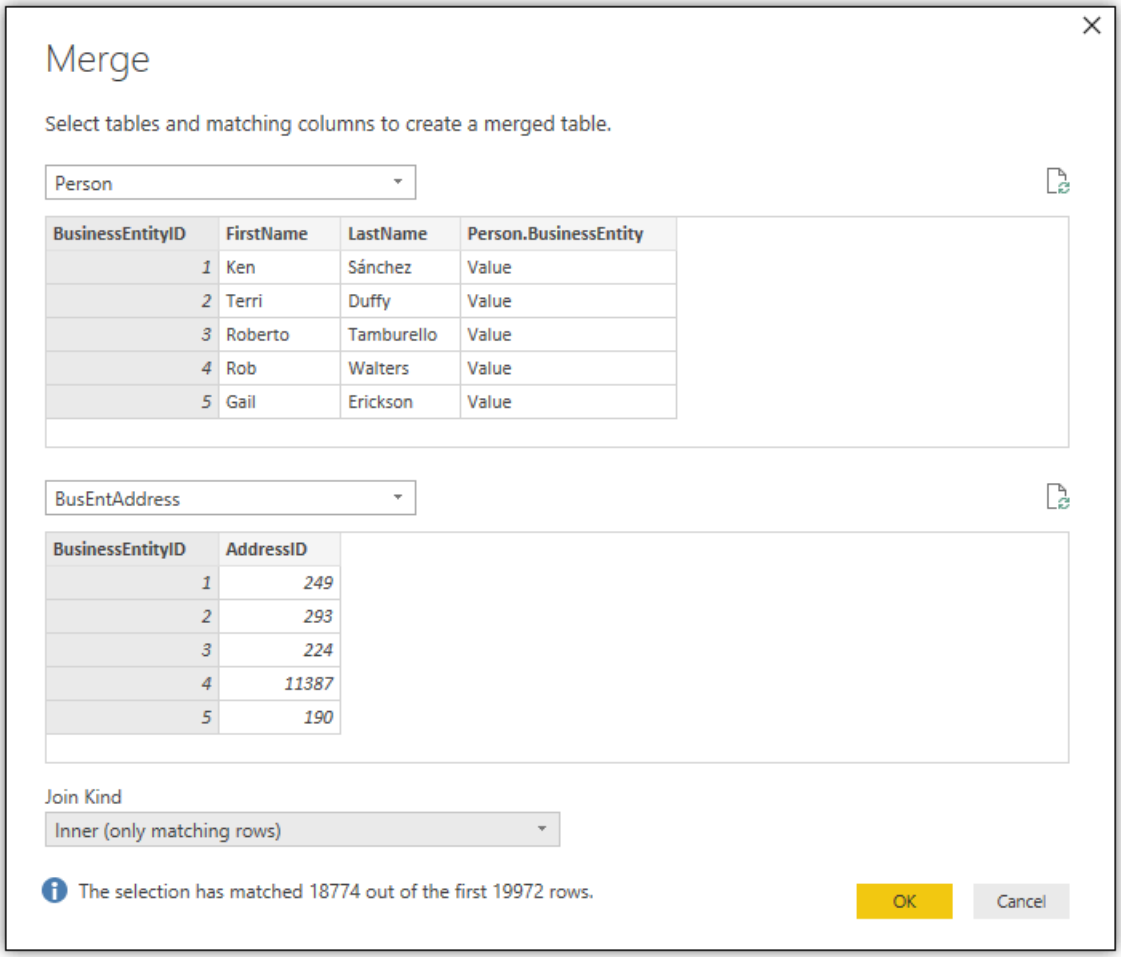


We'll be digging into how to create visualizations in a later article in this series, so I won't be going into any details here. Just know that relationships can make this process simpler because the data is mapped for you.

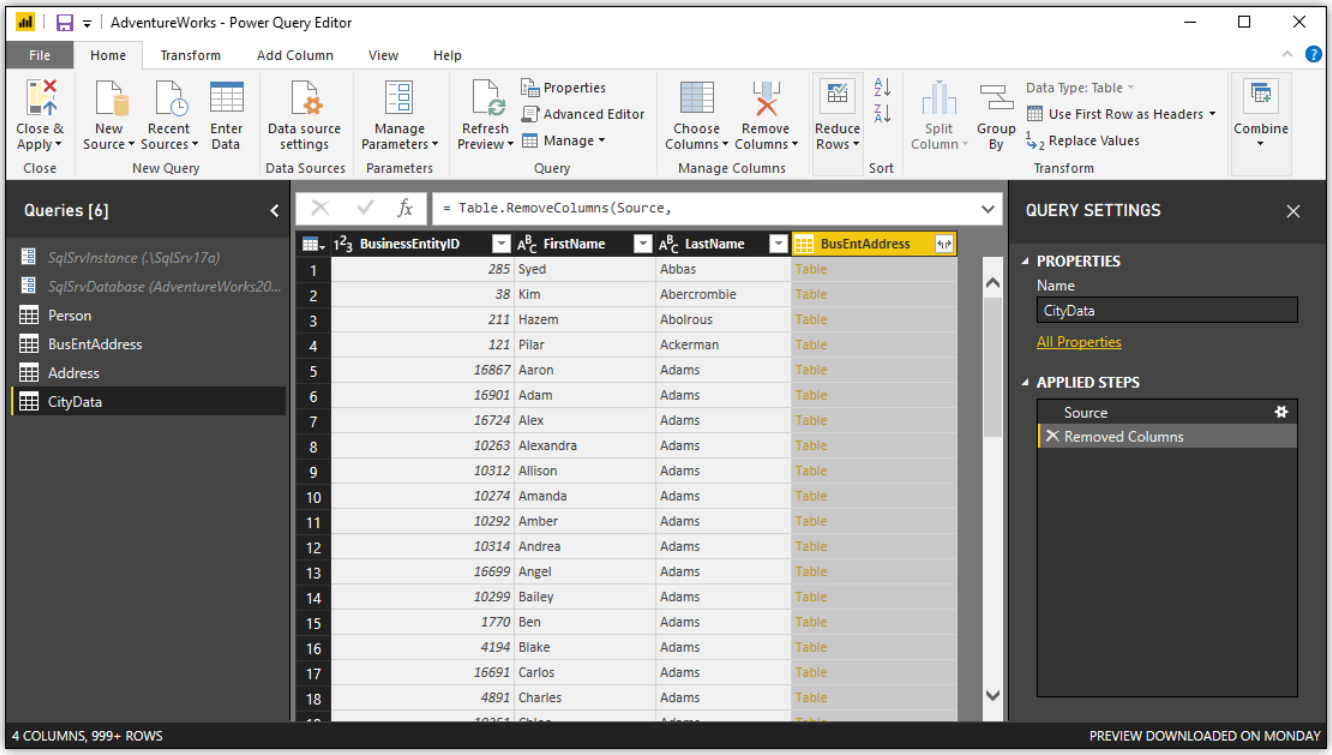
Merging Datasets

In some cases, you might want to combine multiple datasets into a single one. For example, you might decide to merge the three datasets you created earlier. To do so, open Query Editor and select the *Person* query. On the *Home* ribbon, click the *Merge Queries* down arrow and then click *Merge Queries as New*.

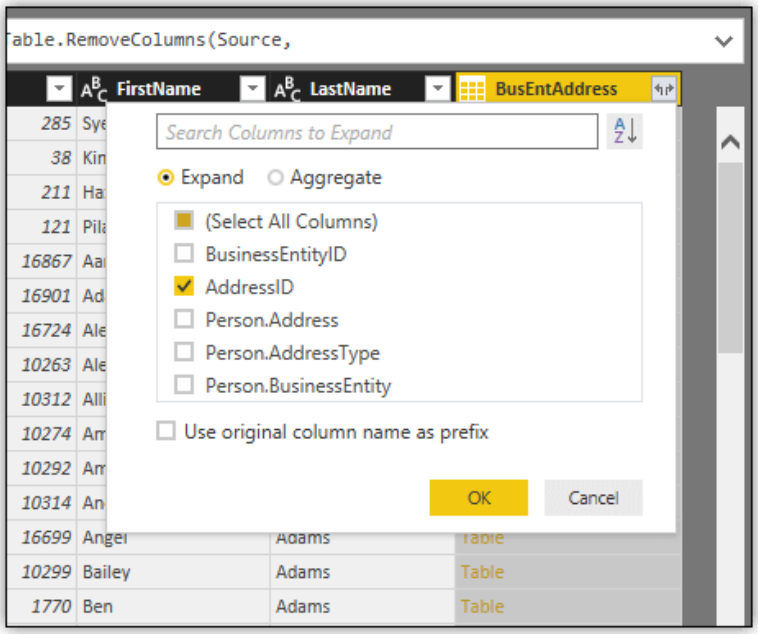
In the *Merge* dialog box (shown in the following figure), verify that the *Person* dataset is selected in the first drop-down list. Next, select *BusEntAddress* from the second (middle) drop-down list, and then select *Inner (only matching rows)* from the *Join Kind* drop-down list. This tells Query Editor to create an inner join between the two queries. Then, for each referenced dataset, select the *BusinessEntityID* column and click *OK*.



When you click *OK,* Query Editor creates a new query, which you can rename *CityData*. The *CityData* query contains the three primary columns from the *Person* dataset, along with a number of relationship columns. Delete all the relationship columns except the last one, *BusEntAddress,* which represents the *BusEntAddress* query. Your merged query should now look like the one shown in the following figure.

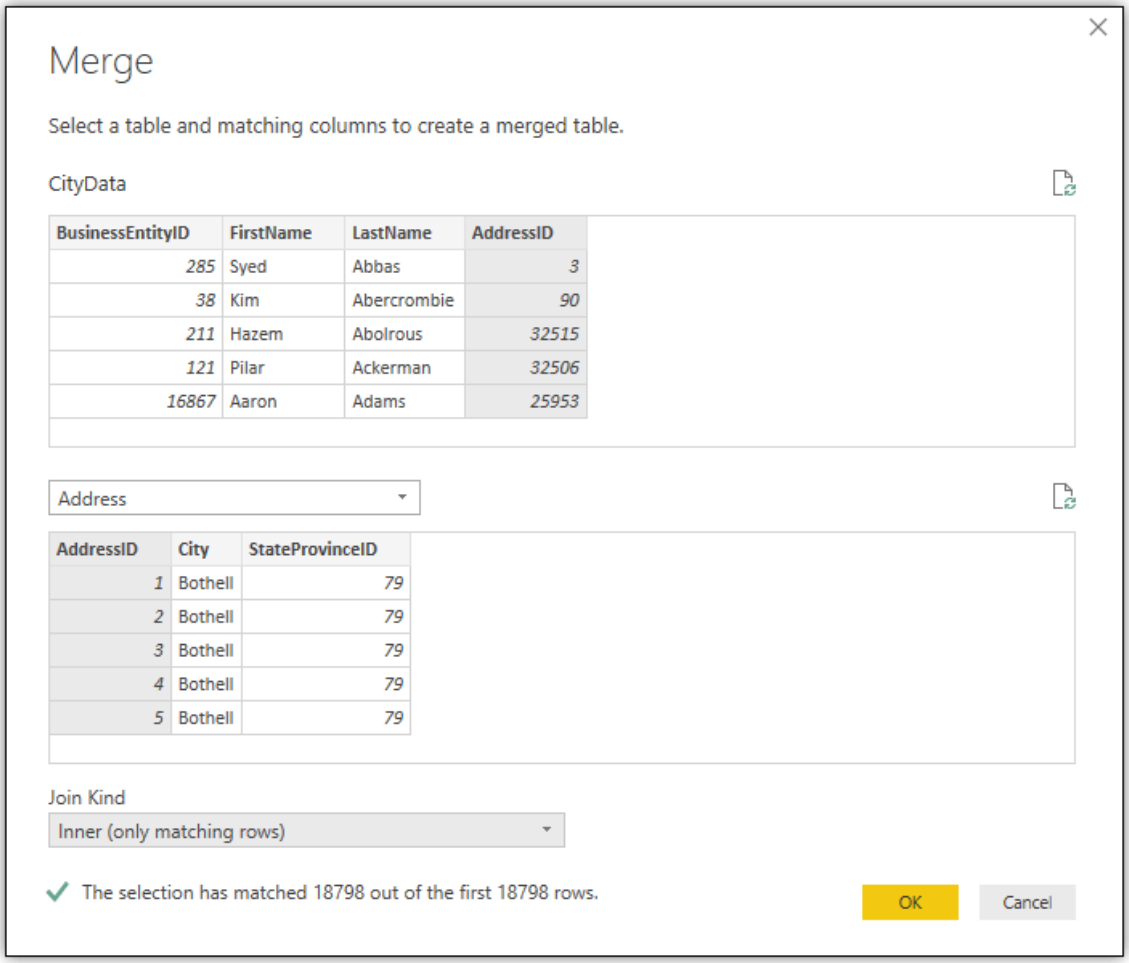


The next step is to select one or more columns from the *BusEntAddress* query to add to the merged query. To select the columns, click the icon at the top right corner of the *BusEntAddress* column and clear all but the *AddressID* column, as shown in the following figure. You do not need to add the *BusinessEntityID* column because it is already included in the *Person* dataset, and the other columns are relationship columns.

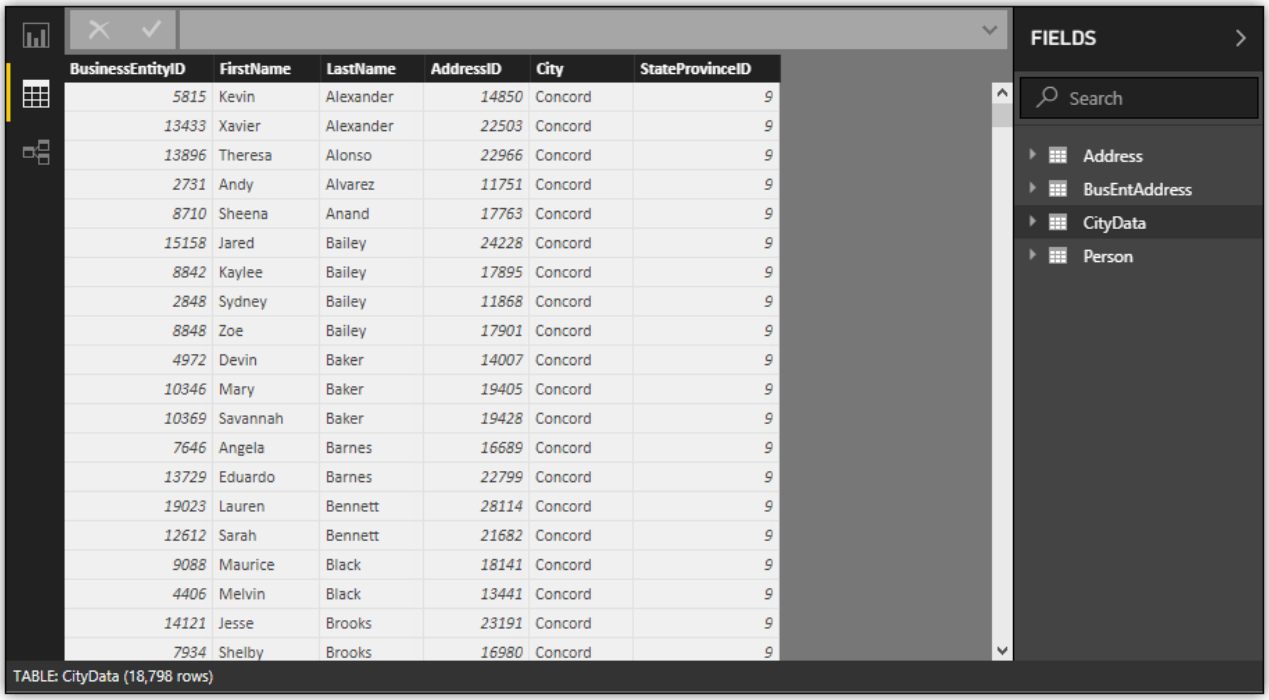


Also clear the *Use original column name as prefix* checkbox to keep the column name simple, and then click *OK*.
The *CityData* query will now include the *AddressID* column.

The next step is to merge the *Address* query into the *CityData* query. With the *CityData* query still selected in Query Editor, click the *Merge Queries* button on the *Home* ribbon. In the *Merge* dialog box (shown in the following figure), select *Address* from the second drop-down list and select *Inner (only matching rows)* from the *Join Kind* drop-down list. For each data set, select the *AddressID* column and then click *OK*.



Next, click the icon at the top right corner of the *Address* reference column and clear all but the *City* and *StateProvinceID* columns. Also clear the *Use original column name as prefix* checkbox, if selected, and then click *OK*. Your merged dataset should now look like the one in the following figure.



BusinessEntityID	FirstName	LastName	AddressID	City	StateProvinceID
5815	Kevin	Alexander	14850	Concord	9
13433	Xavier	Alexander	22503	Concord	9
13896	Theresa	Alonso	22966	Concord	9
2731	Andy	Alvarez	11751	Concord	9
8710	Sheena	Anand	17763	Concord	9
15158	Jared	Bailey	24228	Concord	9
8842	Kaylee	Bailey	17895	Concord	9
2848	Sydney	Bailey	11868	Concord	9
8848	Zoe	Bailey	17901	Concord	9
4972	Devin	Baker	14007	Concord	9
10346	Mary	Baker	19405	Concord	9
10369	Savannah	Baker	19428	Concord	9
7646	Angela	Barnes	16689	Concord	9
13729	Eduardo	Barnes	22799	Concord	9
19023	Lauren	Bennett	28114	Concord	9
12612	Sarah	Bennett	21682	Concord	9
9088	Maurice	Black	18141	Concord	9
4406	Melvin	Black	13441	Concord	9
14121	Jesse	Brooks	23191	Concord	9
7934	Shelby	Brooks	16980	Concord	9

With the merged query in place, you have a single dataset from which you can perform other operations. For example, you can apply additional transformations such as pivoting data, splitting columns, or filtering rows, while still preserving the original datasets. A single dataset can also make it easier create visualizations because you don't have to navigate multiple sources. Like any feature in Power BI Desktop, your specific requirements will determine whether you can benefit from merged queries.

Using T-SQL Queries to Retrieve SQL Server Data

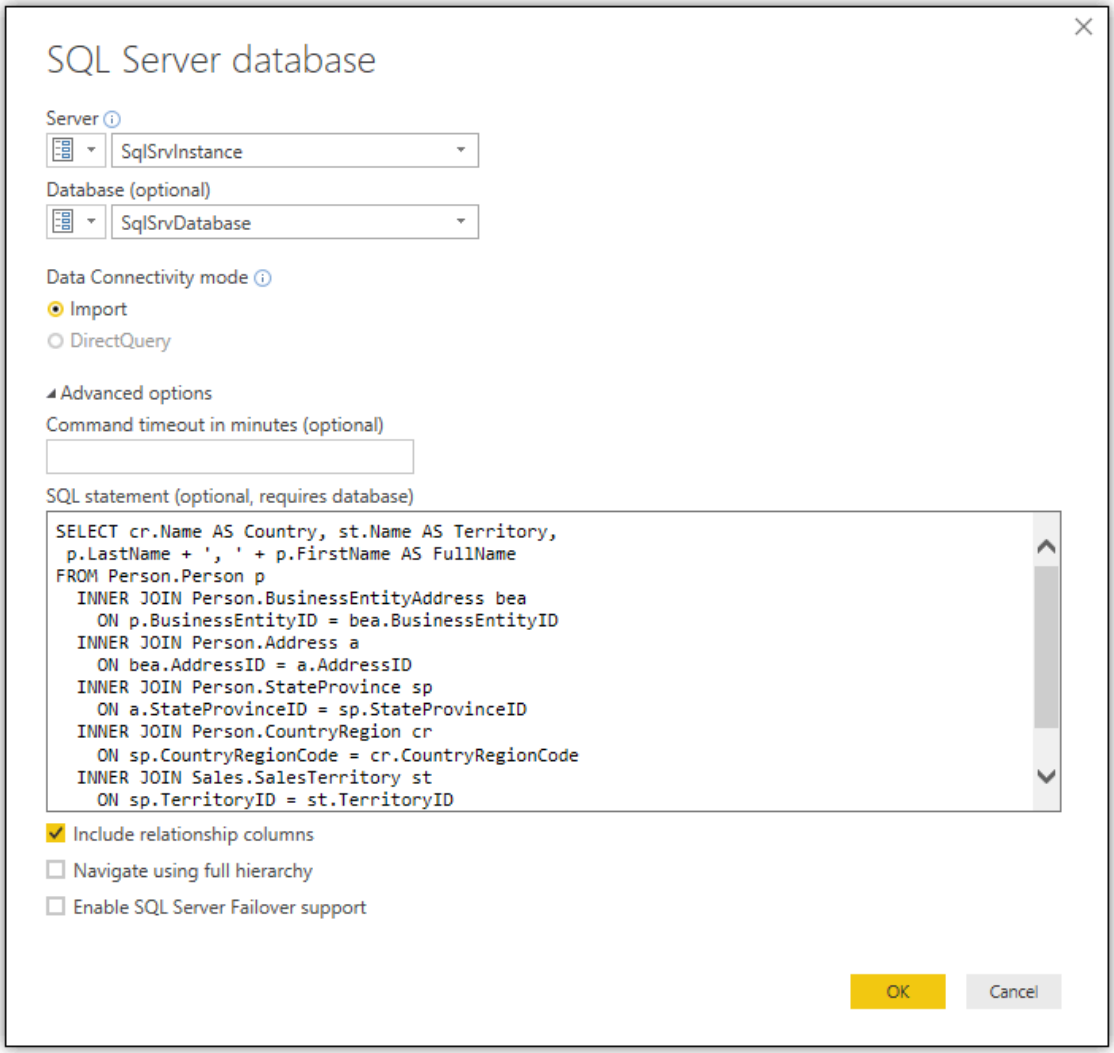
Although you can transform data in a variety of ways in Power BI Desktop, one of the most powerful tools you have at your disposal is the ability to use T-SQL queries to return data from a SQL Server database, rather than specifying individual tables or views. With a query, you can retrieve exactly the data you want, without having to remove columns, filter rows, or take other steps after importing the data. In this way, you're working with a single dataset that includes exactly the data you need, rather than trying to navigate multiple datasets that contain a lot of data you don't want.

In the next example, you'll use the following **SELECT** statement to join together six tables in the **AdventureWorks2017** database and return the data to Power BI Desktop:

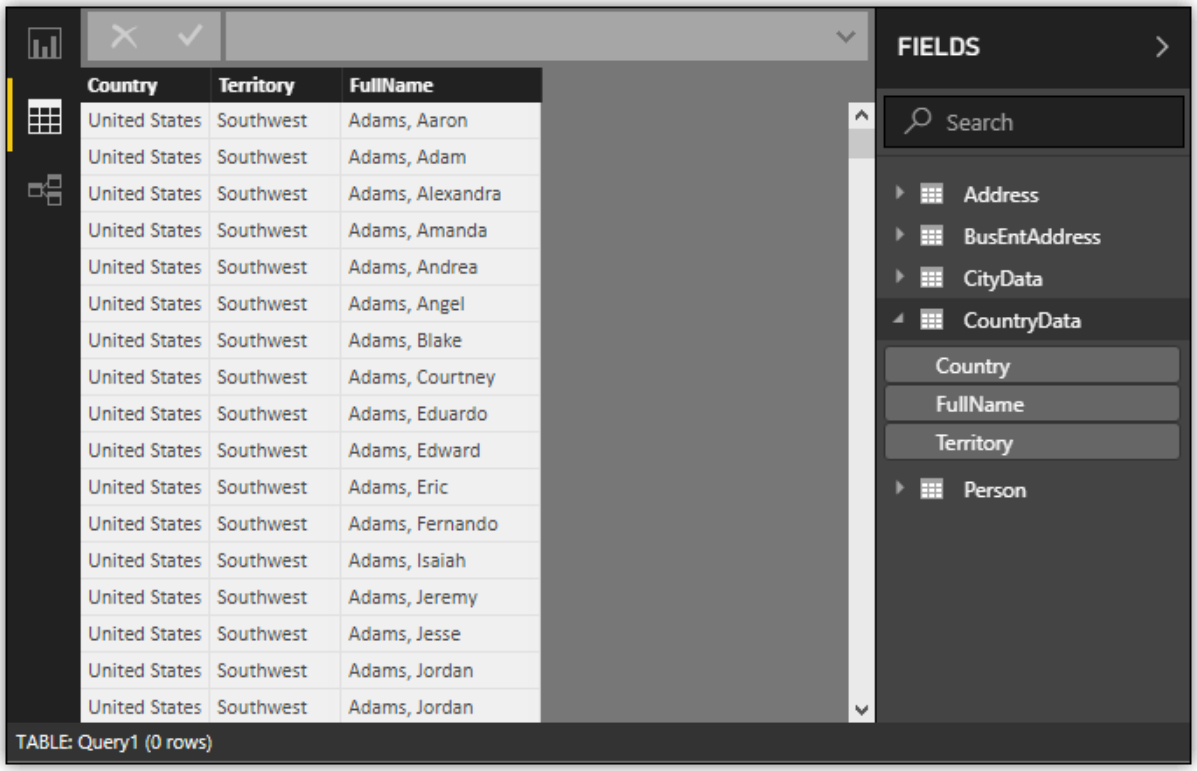
```
SELECT cr.Name AS Country, st.Name AS Territory,
 p.LastName + ', ' + p.FirstName AS FullName
FROM Person.Person p
  INNER JOIN Person.BusinessEntityAddress bea
    ON p.BusinessEntityID = bea.BusinessEntityID
  INNER JOIN Person.Address a
    ON bea.AddressID = a.AddressID
  INNER JOIN Person.StateProvince sp
    ON a.StateProvinceID = sp.StateProvinceID
  INNER JOIN Person.CountryRegion cr
    ON sp.CountryRegionCode = cr.CountryRegionCode
  INNER JOIN Sales.SalesTerritory st
    ON sp.TerritoryID = st.TerritoryID
WHERE PersonType = 'IN' AND st.[Group] = 'North America'
ORDER BY cr.Name, st.Name, p.LastName, p.FirstName;
```

To run the query, click *Get Data* on the *Home* ribbon in the main Power BI Desktop window. When the *Get Data* dialog box appears, navigate to the *Database* category and double-click *SQL Server database*.

In the *SQL Server database* dialog box, provide the name of the SQL Server instance and the target database and then click the *Advanced options* arrow, which expands the dialog box. In the *SQL statement* box, type or paste the above **SELECT** statement, as shown in the following figure.



When you click *OK,* Power BI Desktop displays a preview window, showing you a subset of the data you'll be importing. Click *Load* to import the data into Power BI Desktop. After the data has loaded, go to *Data* view, right-click the new dataset, and then click *Rename*. Type *CountryData* and press *Enter*. The new dataset should look similar to the one shown in the following figure.



Country	Territory	FullName
United States	Southwest	Adams, Aaron
United States	Southwest	Adams, Adam
United States	Southwest	Adams, Alexandra
United States	Southwest	Adams, Amanda
United States	Southwest	Adams, Andrea
United States	Southwest	Adams, Angel
United States	Southwest	Adams, Blake
United States	Southwest	Adams, Courtney
United States	Southwest	Adams, Eduardo
United States	Southwest	Adams, Edward
United States	Southwest	Adams, Eric
United States	Southwest	Adams, Fernando
United States	Southwest	Adams, Isaiah
United States	Southwest	Adams, Jeremy
United States	Southwest	Adams, Jesse
United States	Southwest	Adams, Jordan
United States	Southwest	Adams, Jordan

Now you have the dataset exactly as you need it, without all the extra steps that come with importing individual tables. You can still apply additional transformations if necessary, but often that will not be needed.

In Power BI Desktop, you can run a wide range of T-SQL queries when retrieving data from a SQL Server database, including **EXECUTE** statements that call stored procedures. For example, you can call the **sp_execute_external_script** stored procedure in order to run an R script (SQL Server 2016 or later) or Python script (SQL Server 2017 or later), as shown in the following example:

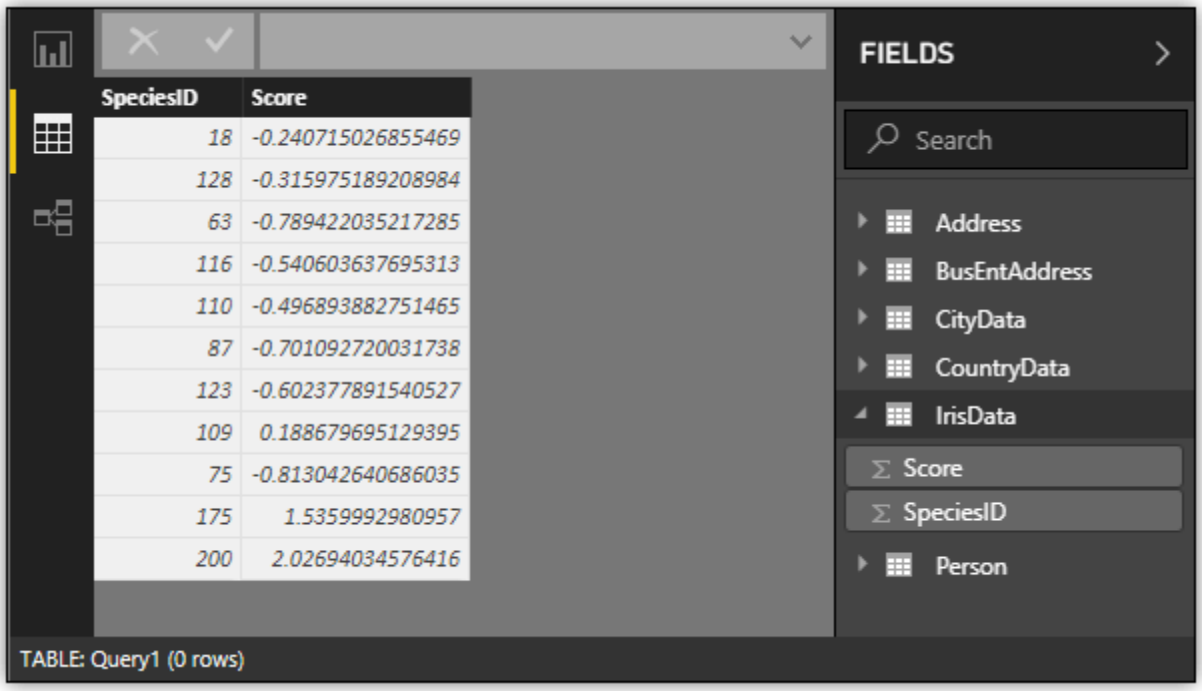
```
DECLARE @pscript NVARCHAR(MAX);
SET @pscript = N'
# import Python modules
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from microsoftml import rx_oneclass_svm, rx_predict
# create data frame
iris = load_iris()
df = pd.DataFrame(iris.data)
df.reset_index(inplace=True)
df.columns = ["Plant", "SepalLength", "SepalWidth", "PetalLength", "PetalWidth"]
# split data frame into training and test data
train, test = train_test_split(df, test_size=.06)
# generate model
svm = rx_oneclass_svm(
  formula="~ SepalLength + SepalWidth + PetalLength + PetalWidth", data=train)
# add anomalies to test data
test2 = pd.DataFrame(data=dict(
Plant = [175, 200],
SepalLength = [2.9, 3.1],
SepalWidth = [.85, 1.1],
PetalLength = [2.6, 2.5],
PetalWidth = [2.7, 3.2]))
test = test.append(test2)
# score test data
scores = rx_predict(svm, data=test, extra_vars_to_write="Plant")
OutputDataSet = scores';
EXEC sp_execute_external_script
  @language = N'Python',
  @script = @pscript
  WITH RESULT SETS(
    (SpeciesID INT, Score FLOAT(25)));
```

NOTE: You need to have Machine Learning Services (In-Database) installed and enable the sp_execute_external_script property to run Python in SQL Server.

In this case, the **sp_execute_external_script** stored procedure is running a Python script that identifies anomalies in a sample dataset containing iris flower data. I based this example on one in a previous article I wrote, SQL Server Machine Learning

Services – Part 4: Finding Data Anomalies with Python, which is part of a series on Python in SQL Server. Refer to that article for a full explanation of how this statement works.

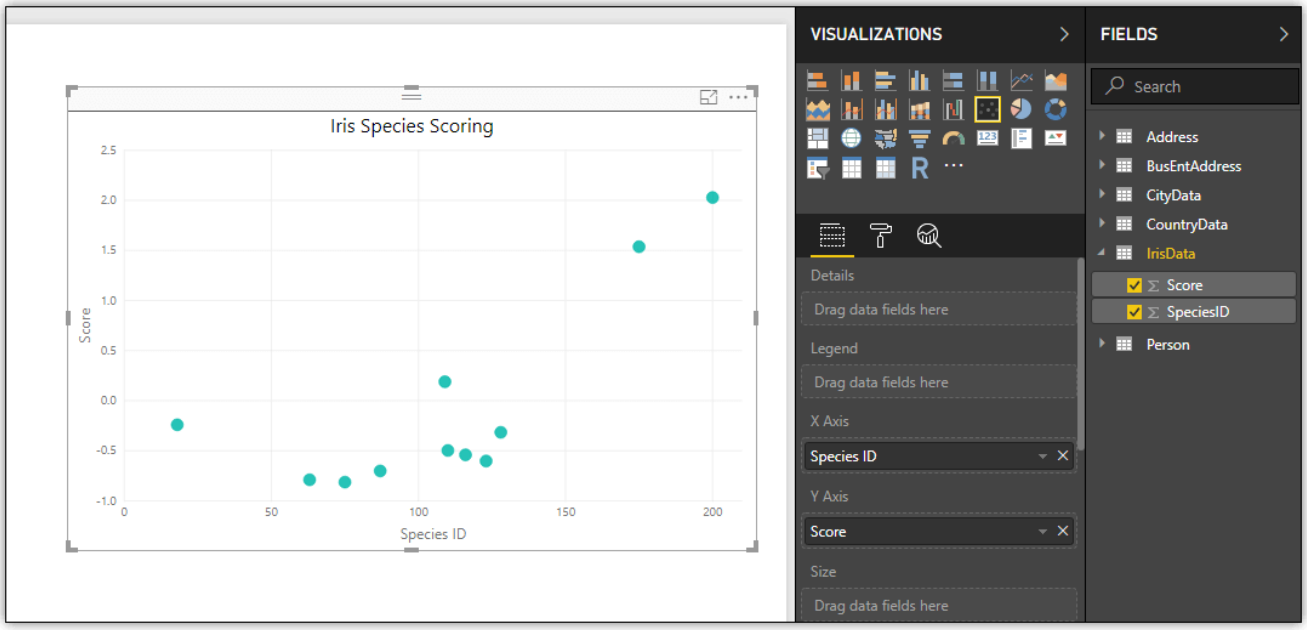
If you run this script, you should end up with the dataset similar to the one shown in the following figure, renamed to *IrisData*. (Because you're working with random data, the values will usually be different each time you run the query.) The dataset includes the IDs used to identify iris species and scores that help to determine anomalies in the data.



SpeciesID	Score
18	-0.240715026855469
128	-0.315975189208984
63	-0.789422035217285
116	-0.540603637695313
110	-0.496893882751465
87	-0.701092720031738
123	-0.602377891540527
109	0.188679695129395
75	-0.813042640686035
175	1.5359992980957
200	2.02694034576416

Although this example does not include SQL Server data, it demonstrates that you can run a Python script from Power BI Desktop. You can also create a script that includes SQL Server data or data from multiple sources, opening up an even wider range of possibilities for pulling data into Power BI Desktop.

Best of all, you can take advantage of Python's analytical capabilities, while importing the data in exactly the format you need, allowing you to easily create visualizations that provide critical insights into the available information. For example, I created a *Scatter chart* visualization based on this *IrisData* dataset, as shown in the following figure. Notice the two points in the chart toward the upper right corner. These represent the anomalies identified by the Python script.



Making the Most of SQL Server Data

The ability to run T-SQL queries in Power BI Desktop gives you a powerful tool for accessing SQL Server data. Not only does this allow you to maximize the benefits of the SQL Server query engine, but also minimize the size of the datasets you import into Power BI Desktop, while reducing the number of transformations you have to perform.

Of course, you must understand how to use T-SQL in order to take this approach, but if you do—or can get someone who does—you'll have a great deal of flexibility for working with SQL Server data. That said, even if you retrieve data on a table-by-table basis, you still have a number of effective tools for working with SQL Server data and preparing it for creating meaningful visualizations that can be posted to the Power BI service.

Although the Power BI service lets you retrieve data from a number of sources, it does not provide a data connector for SQL Server. Of course, you can export SQL Server data to a file and then import that file into the service, but you cannot connect directly from the service to SQL Server, despite the critical role that SQL Server continues to play in many of today's organizations. The better you understand how to import and transform SQL Server data in Power BI Desktop, the more effectively you can visualize SQL Server data in a variety of ways.

Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6

In this article of the Power BI series, Robert Sheldon demonstrates how to work with the Power Query M language to import and transform data.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

In Power BI Desktop, a dataset is defined by a single query that specifies what data to include and how to transform that data. The query is made up of related steps that build on each other to produce the final dataset. Once you've defined the dataset, you can use it to create visualizations that you add to your reports, which you can then publish to the Power BI service.

At the heart of the query is the Power Query M formula language, a case-sensitive, data mashup language available to Power BI Desktop, Power Query in Excel, and the *Get & Transform* import feature in Excel 2016. As with many languages, Power Query statements are made up of individual language elements such as functions, variables, expressions, and primitive and structured values that together define the logic necessary to shape the data.

Each query in Power BI Desktop is a single Power Query **let** expression that contains all the code elements necessary to define a dataset. A **let** expression is made up of a **let** statement and an **in** statement, as shown in the following syntax:

```
let
  <em>variable</em> = <em>expression</em> [,...]
in
  <em>variable</em>
```

The **let** statement includes one or more procedural steps that define the query. Each step is essentially a variable assignment that consists of a variable name and an expression that provides the variable's value. The expression defines the logic for adding, removing, or transforming the data in a specific way. This is where you'll be doing the bulk of your work, as you'll see in the examples to follow.

The variable can be of any supported Power Query type. Each variable within a **let** expression must have a unique name, but Power Query is fairly flexible about what you call it. You can even include spaces in the name, but you have to enclose it in double-quotes and precede it with a hash tag, a somewhat cumbersome naming convention (and one I prefer to avoid). The

variable names are also the same names used to identify steps in the *Applied Steps* section in Query Editor's right pane, so use names that make sense.

You can include as many procedural steps in your **let** statement as necessary and practical. If you include multiple steps, you must use commas to separate them. Each step generally builds on the preceding one, using the variable from that step to define the logic in the new step.

Strictly speaking, you do not have to define your procedural steps in the same physical order as their logical order. For example, you can reference a variable in the first procedural step that you define in the last procedural step. However, this approach can make the code difficult to debug and cause unnecessary confusion. The accepted convention when writing a **let** statement is to keep the physical and logical orders in sync.

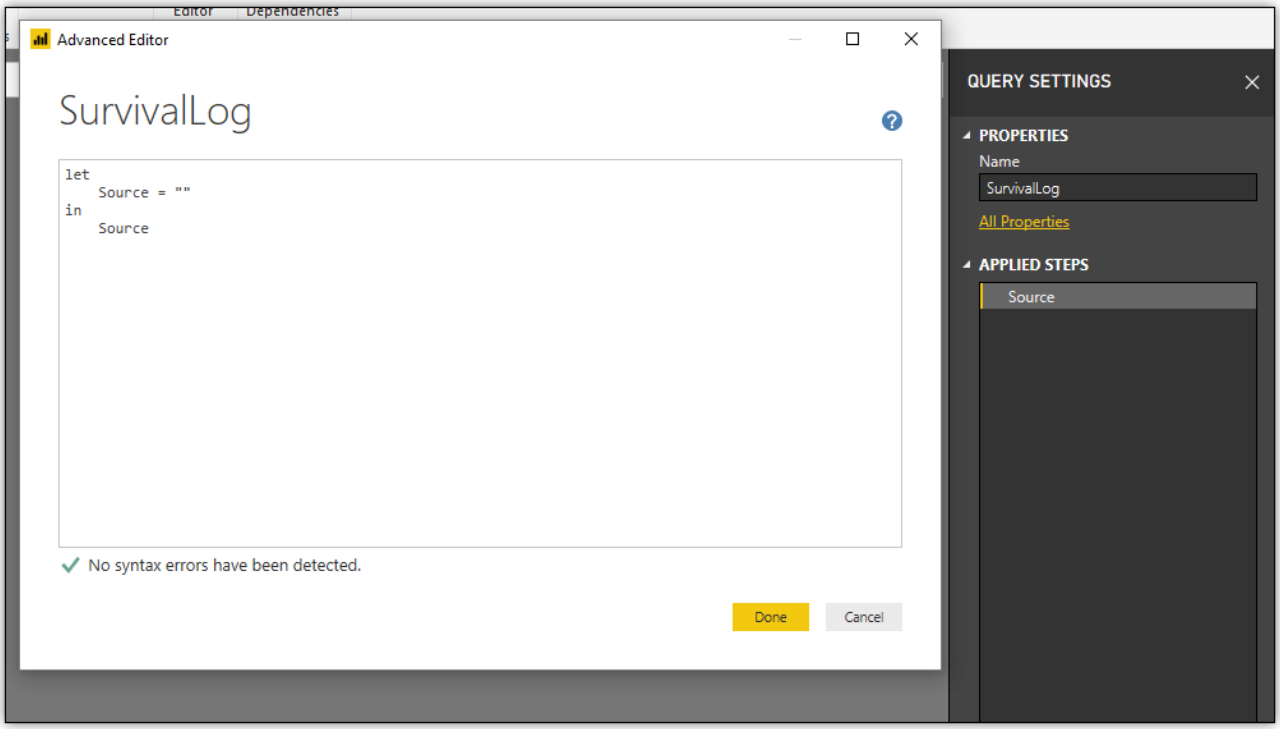
The **in** statement returns a variable value that defines the dataset's final shape. In most cases, if not all, this will be the last variable defined in your **let** statement. You can specify a different variable from the **let** statement, but there's usually little reason to do. If you simply need to see a variable's value at a specific point in time, you can select its related step in the *Applied Steps* section in Query Editor. When you select a step, you are essentially viewing the contents of the associated variable.

Retrieving Data from a CSV File

This article includes a number of examples that demonstrate how to define the procedural steps necessary to build a query. The examples are based on the *titanic* dataset available from this GitHub page. The *titanic* dataset lists the passengers on the infamous Titanic voyage and shows who survived and who did not. If you plan to try these examples for yourself, you need to create the *titanic.csv* file from the *titanic* dataset and save it to a folder that you can access from Power BI Desktop. On my system, I saved the file to my local folder *C:\DataFiles\titanic.csv*. There are several files named 'titanic' on the GitHub page, so be sure to download the Stat2Data Titanic file.

The first step in building your own Power Query script is to add a blank query to Power BI Desktop. To add the query, click *Get Data* on the *Home* ribbon in the main window, navigate to the *Other* section, and double-click *Blank Query.* This launches Query Editor with a new query listed in the *Queries* pane. The query is named *Query1* or a similarly numbered name, depending on what queries you've already created.

To rename the query, right-click the query in the *Queries* pane, click *Rename,* type *SurvivalLog,* and press *Enter*. Then, on the *View* menu, click *Advanced Editor*. The editor will open with a new **let** expression defined, as shown in the following figure.



The **let** statement includes a single procedural step in which the variable is named **Source** and the expression is an empty string, as indicated by the double quotes. Notice that the variable is also listed in the **in** statement and as a step in the *Applied Steps* section of the right pane. For this exercise, you'll be changing the variable name to **GetPassengers**.

The first procedural step that you'll add will retrieve the contents of the *titanic.csv* file and save them to the **GetPassengers** variable. To add the procedure, replace the existing **let** expression with the following expression:

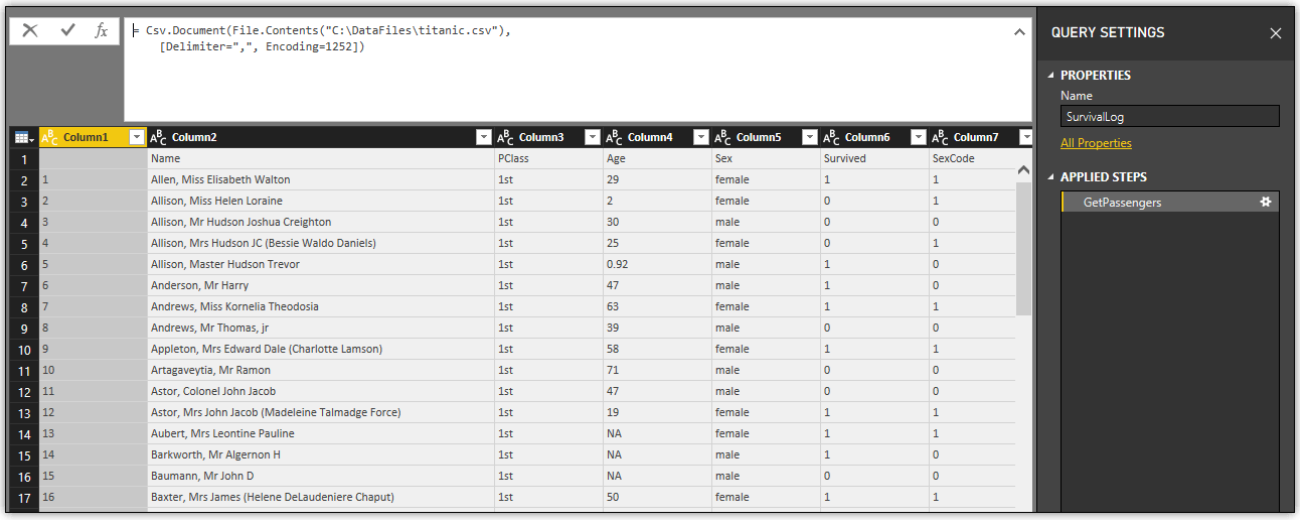
```
let  GetPassengers = Csv.Document(File.Contents("C:\DataFiles\titanic.csv"),     [Delimiter=",", Encoding=1252])in  GetPassengers
```

Notice that the **GetPassengers** variable is specified in the procedural step in the **let** statement and in the **in** statement. In this way, the variable's value will be returned when the **let** expression runs.

The expression in the procedural step includes everything after the equal sign. The expression uses the **Csv.Document** function to retrieve the file's contents as a **Table** object. The function supports several parameters, but only the first one is required to identify the source data. As part of the first parameter, you must also use the **File.Contents** function to return only the document's data.

The second parameter can be specified as a *record* that contains optional settings. In Power Query, a record is a set of fields made up of name/value pairs enclosed in brackets. The record in this example includes the **Delimiter** and **Encoding** options and their values. The **Delimiter** option specifies a comma as the delimiter in the CSV document, and the **Encoding** option specifies the text encoding type of **1252**, which is based on the Windows Western European code page.

That's all there is to setting up a basic **let** expression. After you've entered your code, click *Done* to close the Advanced Editor and run the procedural step. The dataset, as it appears in Query Editor, should look similar to the following figure.



	Column1	Column2	Column3	Column4	Column5	Column6	Column7
1		Name	PClass	Age	Sex	Survived	SexCode
2	1	Allen, Miss Elisabeth Walton	1st	29	female	1	1
3	2	Allison, Miss Helen Loraine	1st	2	female	0	1
4	3	Allison, Mr Hudson Joshua Creighton	1st	30	male	0	0
5	4	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25	female	0	1
6	5	Allison, Master Hudson Trevor	1st	0.92	male	1	0
7	6	Anderson, Mr Harry	1st	47	male	1	0
8	7	Andrews, Miss Kornelia Theodosia	1st	63	female	1	1
9	8	Andrews, Mr Thomas, jr	1st	39	male	0	0
10	9	Appleton, Mrs Edward Dale (Charlotte Lamson)	1st	58	female	1	1
11	10	Artagaveytia, Mr Ramon	1st	71	male	0	0
12	11	Astor, Colonel John Jacob	1st	47	male	0	0
13	12	Astor, Mrs John Jacob (Madeleine Talmadge Force)	1st	19	female	1	1
14	13	Aubert, Mrs Leontine Pauline	1st	NA	female	1	1
15	14	Barkworth, Mr Algernon H	1st	NA	male	1	0
16	15	Baumann, Mr John D	1st	NA	male	0	0
17	16	Baxter, Mrs James (Helene DeLaudeniere Chaput)	1st	50	female	1	1

Notice that the first step in the *Applied Steps* section is named *GetPassengers,* the same as the variable name. The procedural step's expression is also displayed in the small window above the dataset. You can edit the code directly in this window if necessary.

Whenever you add, modify, or delete a step, you should apply and save your changes. A simple way to do this is to click the *Save* icon at the top left corner of the menu bar. When prompted to apply your changes, click *Apply*.

Removing a Column from the Dataset

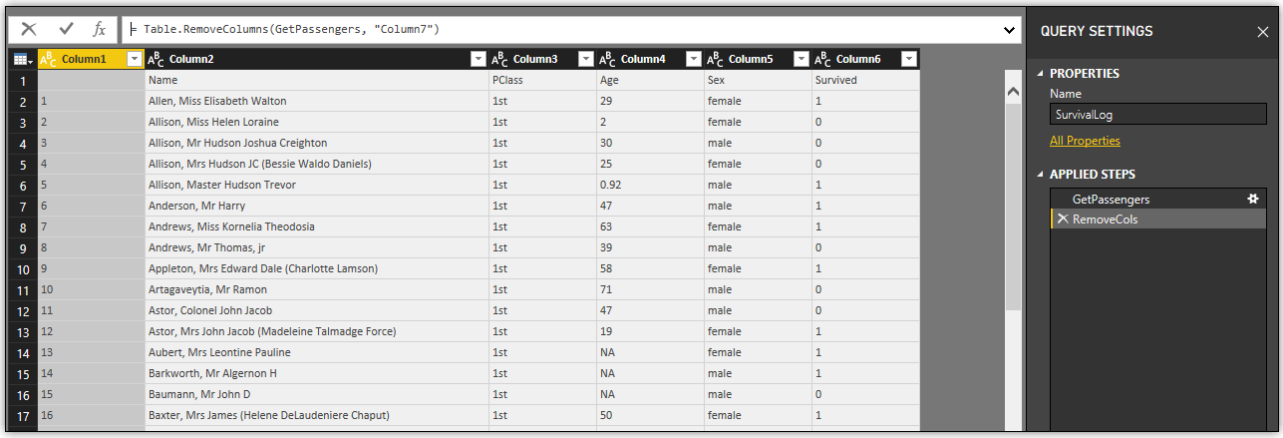
The next procedural step you will add to your **let** statement removes *Column7* from the dataset. Because your dataset is saved as a **Table** object, you can choose from a variety of Power Query **Table** functions to update the data. For this step, you'll use the **Table.RemoveColumns** function to filter out the specified column.

Click *Advanced Editor* to get back to the query. To remove the column, add a comma after the first procedural step and then, on a new line, define the next variable/expression pair, as shown in the following code:

```
let  GetPassengers = Csv.Document(File.Contents("C:\DataFiles\titanic.csv"),     [Delimiter=",", Encoding=1252]),  RemoveCols = Table.RemoveColumns(GetPassengers, "Column7")in  RemoveCols
```

The **Table.RemoveColumns** function requires two parameters. The first parameter specifies the target table that you'll be updating, which in this case, is the **GetPassengers** variable from the previous step. The second argument specifies a single column or a list of columns to be removed. This example specifies only **Column7** from the **GetPassengers** table. The important takeaway here is how the second step builds on the first one and then returns a new result, which is assigned to the **RemoveCols** variable. The new variable contains the same dataset as the **GetPassengers** variable, less the **Column7** data.

After you add the procedural step, replace the **GetPassengers** variable in the **in** statement with the **RemoveCols** variable. Next, click **Done** to close the Advanced Editor, and then save and apply your changes. You dataset should now look like the one shown in the following figure.



	Column1	Column2	Column3	Column4	Column5	Column6
1		Name	PClass	Age	Sex	Survived
2	1	Allen, Miss Elisabeth Walton	1st	29	female	1
3	2	Allison, Miss Helen Loraine	1st	2	female	0
4	3	Allison, Mr Hudson Joshua Creighton	1st	30	male	0
5	4	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25	female	0
6	5	Allison, Master Hudson Trevor	1st	0.92	male	1
7	6	Anderson, Mr Harry	1st	47	male	1
8	7	Andrews, Miss Kornelia Theodosia	1st	63	female	1
9	8	Andrews, Mr Thomas, jr	1st	39	male	0
10	9	Appleton, Mrs Edward Dale (Charlotte Lamson)	1st	58	female	1
11	10	Artagaveytia, Mr Ramon	1st	71	male	0
12	11	Astor, Colonel John Jacob	1st	47	male	0
13	12	Astor, Mrs John Jacob (Madeleine Talmadge Force)	1st	19	female	1
14	13	Aubert, Mrs Leontine Pauline	1st	NA	female	1
15	14	Barkworth, Mr Algernon H	1st	NA	male	1
16	15	Baumann, Mr John D	1st	NA	male	0
17	16	Baxter, Mrs James (Helene DeLaudeniere Chaput)	1st	50	female	1

As you can see, the *RemoveCols* step has been added to the *Applied Steps* section and *Column7* is no longer included in the dataset. To add more procedural steps, follow the same logic as here, continuing to build on your growing foundation.

Promoting the First Row to Headers

The next step is to promote the values in the first row of your dataset to the header row so these values become the column names. To carry out this step, add a comma after the last step you defined and then, on a new line, type the following variable/expression pair:

```
PromoteNames = Table.PromoteHeaders(RemoveCols, [PromoteAllScalars=true])
```

The expression uses the **Table.PromoteHeaders** function to promote the first row to the column headers. The function's first parameter is required and specifies the table to use as the source data (the **RemoveCols** variable). This is just like you saw in the previous example, in which the new step builds on the previous step.

The second parameter, **PromoteAllScalars**, is optional, but is a good one to know about. By default, Power Query promotes only text and number values. When the **PromoteAllScalars** parameter is included and set to **true**, Power Query promotes all scalar values in the first row to headers.

The expression's results are assigned to the **PromoteNames** variable. Be sure to update the **in** statement with this variable name.

Renaming Dataset Columns

Next, you will use the **Table.RenameColumns** function to rename several columns. The function returns a new **Table** object with the column names updated as specified. To rename the columns, add the following procedural step to your **let** statement, being sure to insert a comma after the previous statement:

```
RenameCols = Table.RenameColumns(PromoteNames, {{"", "PsgrID"},
 {"PClass", "PsgrClass"}, {"Sex", "Gender"}})
```

The **Table.RenameColumns** function requires two parameters. The first is the name of the target table (the **PromoteNames** variable from the previous step), and the second is a *list* of old and new column names. In Power Query, a

list is an ordered sequence of values, separated by commas and enclosed in curly brackets. In this example, each value in the list is its own list containing two values: the old column name and the new column name.

Filtering Rows in the Dataset

The next task is to filter out rows that contain **NA** in the **Age** columns and then filter out rows with **Age** values **5** or less. Because the **Age** column was automatically typed as **Text** when imported, you will need to filter the data in three steps, starting with the **NA** values. To filter out these values, add the following variable/expression pair to your code:

```
FilterNA = Table.SelectRows(RenameCols, each [Age] <> "NA")
```

The **Table.SelectRows** function returns a table that contains only the rows that match the defined condition (**Age** not equal to **NA**). As you saw in the previous examples, the function's first argument is target table, in this case, the **RenameCols** variable from the preceding step.

The second argument is an **each** expression specifying that the **Age** value must not equal **NA**. The **each** keyword indicates that the expression should be applied to each row in the target table. As a result, the new table, which is assigned to the **FilterNA** variable, will not include any rows with an **Age** value of **NA**.

Once you've remove the **NA** values, you should convert the **Age** column to the **Number** data type so you can work with the data more effectively, such as being able to filter the data based on a numerical age. When changing the type of the **Age** column, you can also change the type of the **PsgrID** column to the **Int64** type. This way, you're referencing the IDs by integers rather than text. To convert to the new data types, add the following variable/expression pair to the **let** statement:

```
ChangeTypes = Table.TransformColumnTypes(FilterNA,
  {{"PsgrID", Int64.Type}, {"Age", Number.Type}})
```

The expression uses the **Table.TransformColumnTypes** function to change the data types. The first parameter is the name of the target table (**FilterNA**), and the second parameter is a list of the columns to be updated, along with their new data types. Each value is its own list that includes the name of the column and type.

Once the **Age** column has been updated with the **Number** type, you can filter out the ages based on a numerical value, as shown in the following procedural step.

```
FilterKids = Table.SelectRows(ChangeTypes, each [Age] > 5)
```

Your data set should now include only the desired data, as it exists in the **FilterKids** variable.

Replacing Column Values in a Dataset

Another handy transformation you might need to do at times is to replace one column value for another. For this example, you will replace the **0** value in the **Survived** column with **No** and the **1** value with **Yes**, using the **Table.ReplaceValue** function. To replace the **0** values, add the following procedural step to your code:

```
Replace0 = Table.ReplaceValue(FilterKids, "0", "No",
  Replacer.ReplaceText, {"Survived"})
```

The **Table.ReplaceValue** function requires the following five parameters:

- **table:** The target table, in this case, the variable from the preceding step.
- **oldValue:** The original value to be replaced.
- **newValue:** The value that will replace the original value.
- **replacer:** A **Replacer** function that carries out the replacement operation.
- **columnsToSearch:** The column or columns whose values should be replaced.

Most of the parameters should be self-explanatory, except perhaps the **replacer** parameter. For this, you can choose from several functions that work in conjunction with the **Table.ReplaceValue** function to update the values. In this case, the **Replacer.ReplaceText** function is used because text values are being replaced.

After you replace the **0** values, you can replace the **1** values with **Yes**, using the same construction:

```
Replace1 = Table.ReplaceValue(Replace0, "1", "Yes",   Replacer.ReplaceText, {"Survived"})
```

Now the values in the **Survived** column are more readable for those who might not understand the meaning of the **0** and **1** values. It also makes it less likely that the data will cause any confusion.

Changing the Case of Column Values

Another way you can modify column values is to change how they are capitalized. For this example, you will update the values in the **Gender** column so the first letter is capitalized, rather than being all lowercase. To make these changes, add the following procedural step to the **let** statement:

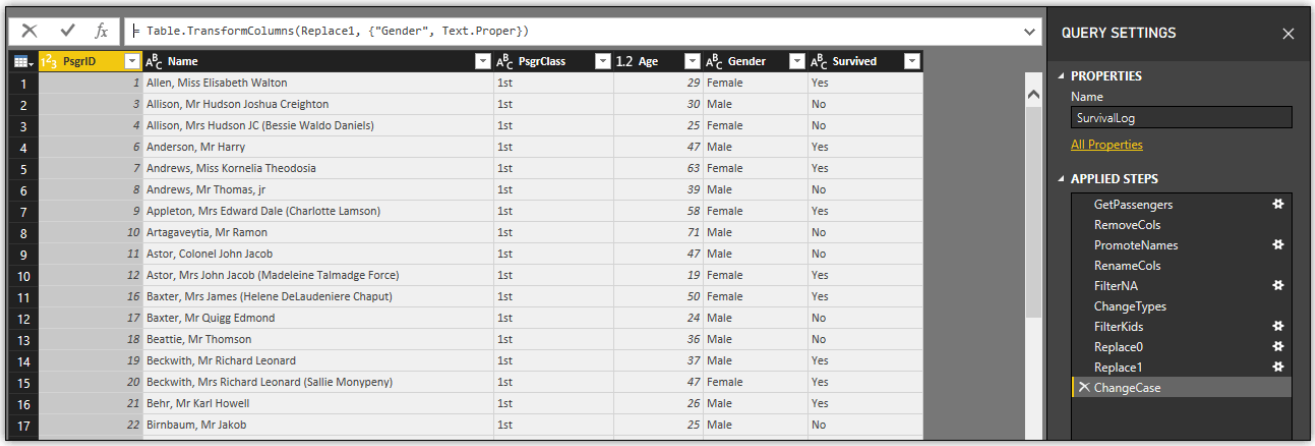
```
ChangeCase = Table.TransformColumns(Replace1, {"Gender", Text.Proper})
```

The expression uses the **Table.TransformColumns** function to update the values. The function requires two parameters. The first is the table to be updated, and the second is a list of operations to carry out. For each operation, you must provide the target column and an expression that carries out the operation. (At times, you might need to provide additional information.) For this example, there is only one operation, so you need only specify the **Gender** column and the **Text.Proper** function, enclosed in curly braces. The function converts the first letter in each word in the **Gender** column to uppercase.

At this point, your **let** expression, in its entirety, should look similar to the following code:

```
let
 GetPassengers = Csv.Document(File.Contents("C:\DataFiles\titanic.csv"),
  [Delimiter=",", Encoding=1252]),
 RemoveCols = Table.RemoveColumns(GetPassengers, "Column7"),
 PromoteNames = Table.PromoteHeaders(RemoveCols, [PromoteAllScalars=true]),
 RenameCols = Table.RenameColumns(PromoteNames, {{"", "PsgrID"},
  {"PClass", "PsgrClass"}, {"Sex", "Gender"}}),
 FilterNA = Table.SelectRows(RenameCols, each [Age] <> "NA"),
 ChangeTypes = Table.TransformColumnTypes(FilterNA,
  {{"PsgrID", Int64.Type}, {"Age", Number.Type}}),
 FilterKids = Table.SelectRows(ChangeTypes, each [Age] > 5),
 Replace0 = Table.ReplaceValue(FilterKids, "0", "No",
   Replacer.ReplaceText, {"Survived"}),  Replace1 = Table.ReplaceValue(Replace0, "1", "Yes",     Replacer.ReplaceText, {"Survived"}),  ChangeCase = Table.TransformColumns(Replace1, {"Gender",
Text.Proper})
in
 ChangeCase
```

The **let** statement includes all the steps that makes up your query to this point, with each step building on the preceding step. The following figure shows how the dataset should now appear in Query Editor.



	PsgrID	Name	PsgrClass	Age	Gender	Survived
1	1	Allen, Miss Elisabeth Walton	1st	29	Female	Yes
2	3	Allison, Mr Hudson Joshua Creighton	1st	30	Male	No
3	4	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25	Female	No
4	6	Anderson, Mr Harry	1st	47	Male	Yes
5	7	Andrews, Miss Kornelia Theodosia	1st	63	Female	Yes
6	8	Andrews, Mr Thomas, jr	1st	39	Male	No
7	9	Appleton, Mrs Edward Dale (Charlotte Lamson)	1st	58	Female	Yes
8	10	Artagaveytia, Mr Ramon	1st	71	Male	No
9	11	Astor, Colonel John Jacob	1st	47	Male	No
10	12	Astor, Mrs John Jacob (Madeleine Talmadge Force)	1st	19	Female	Yes
11	16	Baxter, Mrs James (Helene DeLaudeniere Chaput)	1st	50	Female	Yes
12	17	Baxter, Mr Quigg Edmond	1st	24	Male	No
13	18	Beattie, Mr Thomson	1st	36	Male	No
14	19	Beckwith, Mr Richard Leonard	1st	37	Male	Yes
15	20	Beckwith, Mrs Richard Leonard (Sallie Monypeny)	1st	47	Female	Yes
16	21	Behr, Mr Karl Howell	1st	26	Male	Yes
17	22	Birnbaum, Mr Jakob	1st	25	Male	No

Notice that the *Applied Steps* section includes one step for each variable defined in the **let** statement, specified in the same order they appear in the statement. If you select a step, the data displayed in the main windows will reflect the contents of the related variable.

Adding a Calculated Column to a Dataset

Up to this point, each procedural step in your **let** statement represents a discrete action that is built on the preceding step. In some cases, however, you might need to structure your steps in a way that is not quite so linear. For example, you might want to create a calculated column that shows the difference between a passenger's age and the average age for that person's gender. The following steps outline one approach you can take to this column:

1. Calculate the average age of the female passengers and save it to a variable.
2. Calculate the average age of the male passengers and save it to a variable.
3. Add a column that uses the two variables to calculate the age differences.
4. Round the age differences to a more readable number of decimals. (This step is optional.)

To carry out these steps, add the following procedural steps to your **let** statement:

```
                // add calculated column based on average ages
    Female = Table.SelectRows(ChangeCase, each [Gender] = "Female"),
    AvgFemale = List.Average(Table.Column(Female, "Age")),
    Male = Table.SelectRows(ChangeCase, each [Gender] = "Male"),
    AvgMale = List.Average(Table.Column(Male, "Age")),
AddCol = Table.AddColumn(ChangeCase, "AgeDiff", each   if [Gender] = "Female" then [Age] - AvgFemale  else [Age] - AvgMale),RoundDiff = Table.TransformColumns(AddCol,   {"AgeDiff", each
Number.Round(_, 2)})
```

I'll walk you through these steps in smaller chunks so you can better understand how they work. The first line, which is preceded by two forward slashes, indicates a single-line comment. Anything after the slashes to the end of the line is informational only and does not get processed. Power Query also supports multi-line comments that begin with /* and end with */, just like multi-line comments in SQL Server.

It's up to you whether you want to include comments. Whether or not you do, the next step is to calculate the average age of the female passengers. The following two procedural steps give you the value of the age of the average female.

```
Female = Table.SelectRows(ChangeCase, each [Gender] = "Female"),
AvgFemale = List.Average(Table.Column(Female, "Age")),
```

The first step uses the **Table.SelectRows** function to generate a table that includes only rows with a **Gender** value of **Female**. The function is being used here in the same way you saw earlier, except that different data is being filtered out and the results are being saved to the **Female** variable. Notice that the source table for the function is the **ChangeCase** variable from the preceding step.

The second step uses the **List.Average** function to calculate that average for the **Age** values in the **Female** table. The expression returns a scalar value, which is saved to the **AvgFemale** variable. The function requires only one parameter. The parameter includes a second function, **Table.Column**, which passes the values from the **Age** column to the **List.Average** function.

The next step is to find the average age of the male passengers. For this, you can use the same construction, with a few minor changes:

```
Male = Table.SelectRows(ChangeCase, each [Gender] = "Male"),
AvgMale = List.Average(Table.Column(Male, "Age")),
```

Notice that you must again use the **ChangeCase** variable for the source table when calling the **Table.SelectRows** function, even though it is no longer the step directly preceding the new one. You can, in fact, use any of the previous variables in your expressions, as long as it makes sense to do so.

With the **AvgFemale** and **AvgMale** variables in place, you can now add the column, using the **Table.AddColumn** function:

```
AddCol = Table.AddColumn(ChangeCase, "AgeDiff", each   if [Gender] = "Female" then [Age] - AvgFemale  else [Age] - AvgMale),
```

The **Table.AddColumn** function requires three parameters. The first is the target table, which is again **ChangeCase**, and the second is the name of the new column, **AgeDiff**.

The third argument is the expression used to generate the column's values. The expression starts with the **each** keyword, which iterates through each row in the target table. This is followed by an **if…then…else** expression that calculates the row's value based on whether the passenger is male or female. If the **Gender** value equals **Female**, then the **AgeDiff** value is set to the **Age** value minus the **AvgFemale** value, otherwise set the **AgeDiff** value is set to the **Age** value minus the **AvgMale** value.

After defining the new column, the final procedural step rounds the **AgeDiff** values to two decimal points:

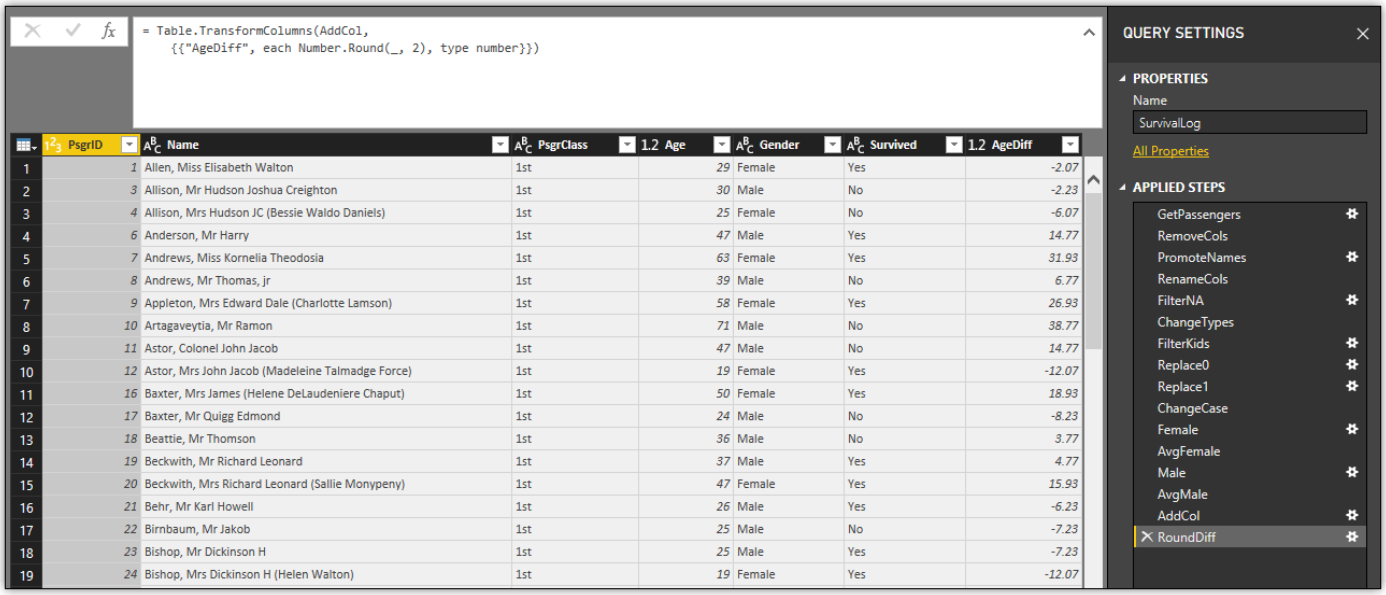
```
RoundDiff = Table.TransformColumns(AddCol,   {"AgeDiff", each Number.Round(_, 2)})
```

The expression includes the **Table.TransformColumns** function, which you saw earlier, but uses the **Number.Round** function to round the values, rather than changing their case. The **Number.Round** function takes two parameters. The first value is an underscore, which represents the column's current value. The second value, **2**, indicates that the value should be rounded to two decimal places.

That completes the steps necessary to create the calculated column. Your **let** expression, in its entirety, should now look like the following code:

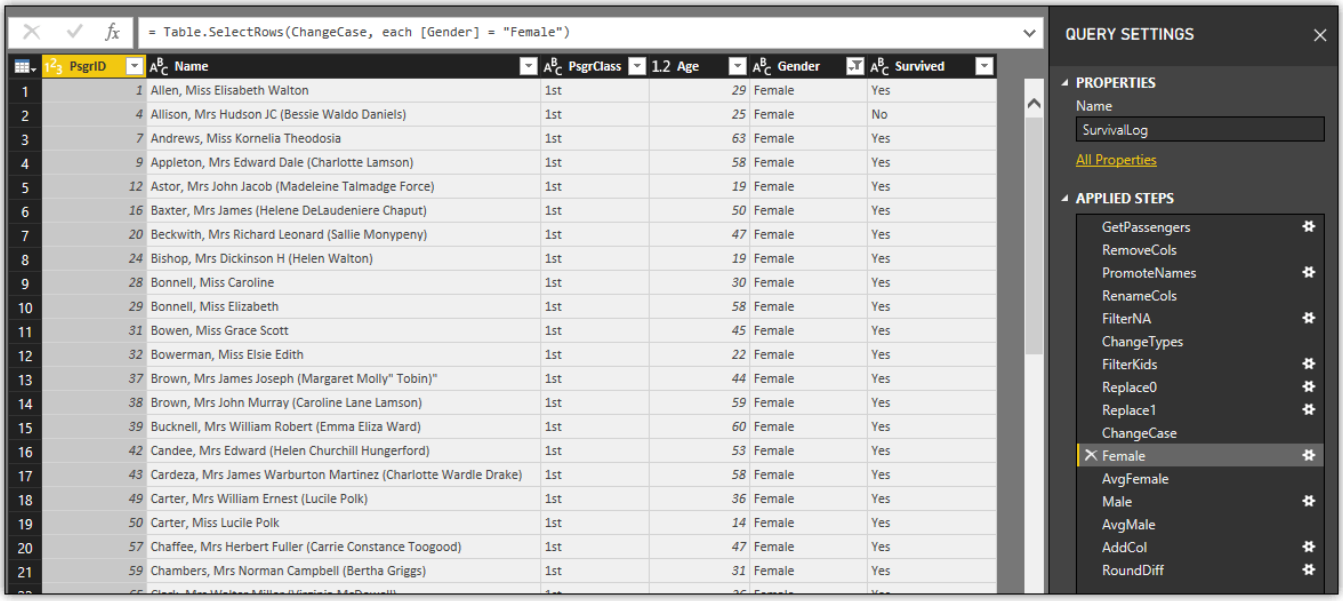
```
let
GetPassengers = Csv.Document(File.Contents("C:\DataFiles\titanic.csv"),
  [Delimiter=",", Encoding=1252]),
RemoveCols = Table.RemoveColumns(GetPassengers, "Column7"),
PromoteNames = Table.PromoteHeaders(RemoveCols, [PromoteAllScalars=true]),
RenameCols = Table.RenameColumns(PromoteNames, {{"", "PsgrID"},
  {"PClass", "PsgrClass"}, {"Sex", "Gender"}}),
FilterNA = Table.SelectRows(RenameCols, each [Age] <> "NA"),
ChangeTypes = Table.TransformColumnTypes(FilterNA,
  {{"PsgrID", Int64.Type}, {"Age", Number.Type}}),
FilterKids = Table.SelectRows(ChangeTypes, each [Age] > 5),
Replace0 = Table.ReplaceValue(FilterKids, "0", "No",
  Replacer.ReplaceText, {"Survived"}),
Replace1 = Table.ReplaceValue(Replace0, "1", "Yes",
  Replacer.ReplaceText, {"Survived"}),
ChangeCase = Table.TransformColumns(Replace1, {"Gender", Text.Proper}),
// add calculated column based on average ages
Female = Table.SelectRows(ChangeCase, each [Gender] = "Female"),
AvgFemale = List.Average(Table.Column(Female, "Age")),
Male = Table.SelectRows(ChangeCase, each [Gender] = "Male"),
AvgMale = List.Average(Table.Column(Male, "Age")),
AddCol = Table.AddColumn(ChangeCase, "AgeDiff", each    if [Gender] = "Female" then [Age] - AvgFemale    else [Age] - AvgMale),  RoundDiff = Table.TransformColumns(AddCol,    {"AgeDiff", each
Number.Round(_, 2)})in  RoundDiff
```

The following figure shows what your dataset should now look like in Query Editor.



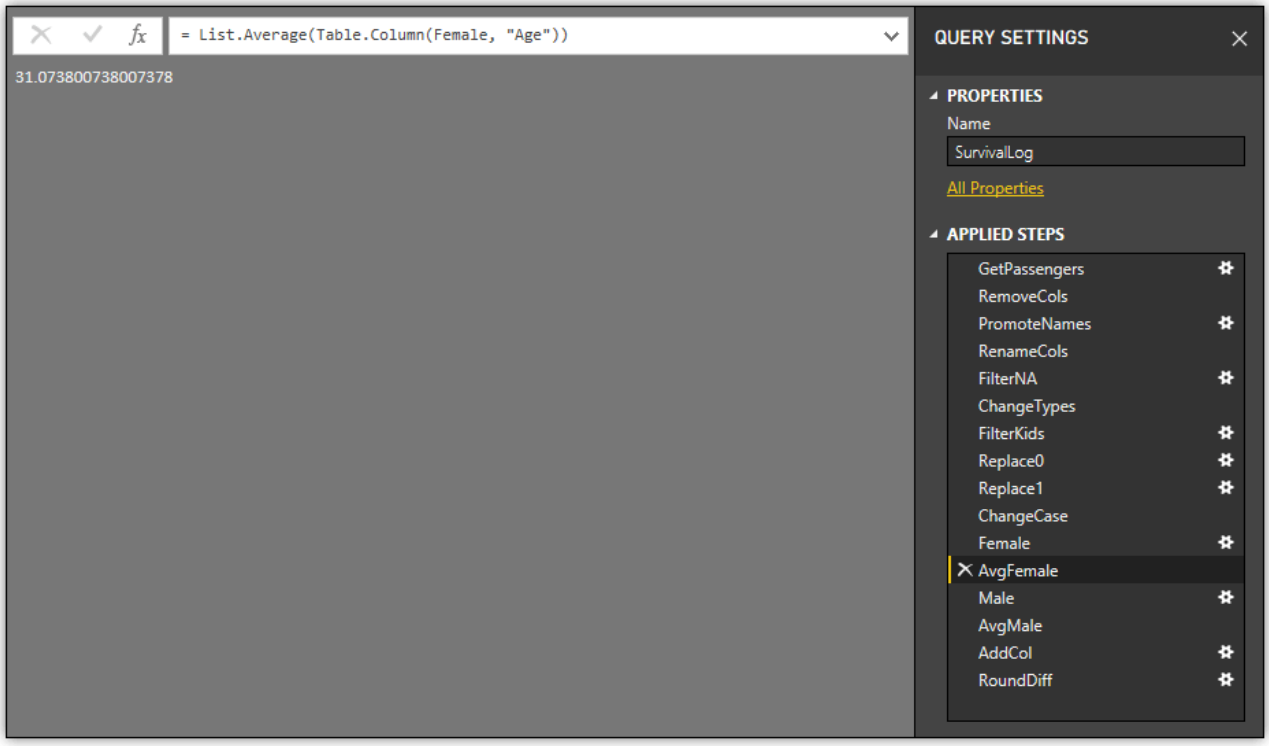
	PsgrID	Name	PsgrClass	Age	Gender	Survived	AgeDiff
1	1	Allen, Miss Elisabeth Walton	1st	29	Female	Yes	-2.07
2	3	Allison, Mr Hudson Joshua Creighton	1st	30	Male	No	-2.23
3	4	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25	Female	No	-6.07
4	6	Anderson, Mr Harry	1st	47	Male	Yes	14.77
5	7	Andrews, Miss Kornelia Theodosia	1st	63	Female	Yes	31.93
6	8	Andrews, Mr Thomas, jr	1st	39	Male	No	6.77
7	9	Appleton, Mrs Edward Dale (Charlotte Lamson)	1st	58	Female	Yes	26.93
8	10	Artagaveytia, Mr Ramon	1st	71	Male	No	38.77
9	11	Astor, Colonel John Jacob	1st	47	Male	No	14.77
10	12	Astor, Mrs John Jacob (Madeleine Talmadge Force)	1st	19	Female	Yes	-12.07
11	16	Baxter, Mrs James (Helene DeLaudeniere Chaput)	1st	50	Female	Yes	18.93
12	17	Baxter, Mr Quigg Edmond	1st	24	Male	No	-8.23
13	18	Beattie, Mr Thomson	1st	36	Male	No	3.77
14	19	Beckwith, Mr Richard Leonard	1st	37	Male	Yes	4.77
15	20	Beckwith, Mrs Richard Leonard (Sallie Monypeny)	1st	47	Female	Yes	15.93
16	21	Behr, Mr Karl Howell	1st	26	Male	Yes	-6.23
17	22	Birnbaum, Mr Jakob	1st	25	Male	No	-7.23
18	23	Bishop, Mr Dickinson H	1st	25	Male	Yes	-7.23
19	24	Bishop, Mrs Dickinson H (Helen Walton)	1st	19	Female	Yes	-12.07

Notice that the *Applied Steps* section includes the variables used to find the average ages, even though the **AddCol** step in the code builds on the **ChangeCase** variable. You can view the contents of these variables by selecting the step in the *Applied Steps* section. For example, the following figure shows the data saved to the **Female** variable.



	PsgrID	Name	PsgrClass	Age	Gender	Survived
1	1	Allen, Miss Elisabeth Walton	1st	29	Female	Yes
2	4	Allison, Mrs Hudson JC (Bessie Waldo Daniels)	1st	25	Female	No
3	7	Andrews, Miss Kornelia Theodosia	1st	63	Female	Yes
4	9	Appleton, Mrs Edward Dale (Charlotte Lamson)	1st	58	Female	Yes
5	12	Astor, Mrs John Jacob (Madeleine Talmadge Force)	1st	19	Female	Yes
6	16	Baxter, Mrs James (Helene DeLaudeniere Chaput)	1st	50	Female	Yes
7	20	Beckwith, Mrs Richard Leonard (Sallie Monypeny)	1st	47	Female	Yes
8	24	Bishop, Mrs Dickinson H (Helen Walton)	1st	19	Female	Yes
9	28	Bonnell, Miss Caroline	1st	30	Female	Yes
10	29	Bonnell, Miss Elizabeth	1st	58	Female	Yes
11	31	Bowen, Miss Grace Scott	1st	45	Female	Yes
12	32	Bowerman, Miss Elsie Edith	1st	22	Female	Yes
13	37	Brown, Mrs James Joseph (Margaret Molly" Tobin)"	1st	44	Female	Yes
14	38	Brown, Mrs John Murray (Caroline Lane Lamson)	1st	59	Female	Yes
15	39	Bucknell, Mrs William Robert (Emma Eliza Ward)	1st	60	Female	Yes
16	42	Candee, Mrs Edward (Helen Churchill Hungerford)	1st	53	Female	Yes
17	43	Cardeza, Mrs James Warburton Martinez (Charlotte Wardle Drake)	1st	58	Female	Yes
18	49	Carter, Mrs William Ernest (Lucile Polk)	1st	36	Female	Yes
19	50	Carter, Miss Lucile Polk	1st	14	Female	Yes
20	57	Chaffee, Mrs Herbert Fuller (Carrie Constance Toogood)	1st	47	Female	Yes
21	59	Chambers, Mrs Norman Campbell (Bertha Griggs)	1st	31	Female	Yes

If you select one of the variables used to store the actual average, Query Editor will display only its scalar value, as shown in the following figure.



Now Query Editor shows only the contents of the **AvgFemale** variable, a rather lengthy number. You can see why I included the final procedural step to round the values in the calculated column.

Navigating the Power Query M Formula Language

Not surprisingly, there's much more to Power Query that what was covered here, but this should give you the foundation you need to start building your own queries in Power BI Desktop. It should also give you a better understanding of how queries are constructed when using point-and-click features to import and transform your data. In this way, you can examine the code to better to understand why you might not be getting the results you expect. You can also start building your query by using point-

and-click operations and then use the Advanced Editor to fine-tune your datasets or introduce logic not easily achieved through the interface.

To make the most of Power Query in Power BI Desktop, you'll need to dig a lot deeper into the language's various elements than what we've covered here, especially the built-in functions. Keep in mind, though, that Power Query was built with Excel in mind, so you might run into elements that don't transfer easily to the Power BI Desktop environment. Even so, the basic principles that define the language and its syntax are the same, and the better you understand them, the more powerful a tool you have for working with data in Power BI Desktop.

Power BI Introduction: Building Reports in Power BI Desktop — Part 7

Power BI contains rich visualizations that allow anyone to build insightful dashboards. In this article, Robert Sheldon shows you how to create Power BI charts, tables, maps, and slicers,

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Until now, this series has focused primarily on building datasets in Power BI Desktop, with the understanding that the data would ultimately be used to create visualizations that provide insights into the underlying information. The visualizations would be included in one or more reports that could then be published to the Power BI service for wider distribution.

This article switches focus from building datasets to creating reports that rely on those datasets, using the tools within Power BI Desktop for visualizing the data. The article offers several examples of how to add visualizations to a report to provide different perspectives of the data. The visualizations are based on a single dataset, created from data in the **AdventureWorks2017** sample database in SQL Server 2017.

If you plan to try out the examples for yourself, you can use the following T-SQL statement to add a dataset to Power BI Desktop:

```
SELECT h.SalesPersonID AS RepID,
  CONCAT(p.LastName, ', ', p.FirstName) AS FullName,
  a.City, sp.Name AS StateProvince, cr.Name AS Country,
  h.OrderDate, h.SubTotal
FROM Sales.SalesOrderHeader h
  INNER JOIN Person.Person p
    ON h.SalesPersonID = p.BusinessEntityID
  INNER JOIN Person.BusinessEntityAddress ba
    ON p.BusinessEntityID = ba.BusinessEntityID
  INNER JOIN Person.Address a
    ON ba.AddressID = a.AddressID
  INNER JOIN Person.StateProvince sp
    ON a.StateProvinceID = sp.StateProvinceID
  INNER JOIN Person.CountryRegion cr
    ON sp.CountryRegionCode = cr.CountryRegionCode
WHERE h.SalesPersonID IS NOT NULL
ORDER BY FullName ASC;
```

For information about using a T-SQL query to create a dataset, refer to Part 5 of this series.

After you import the data into Power BI Desktop, change the name of the dataset to *Sales,* update the format of the *OrderDate* column to show only the dates (not the times), and update the *SubTotal* column to show whole number currency values.

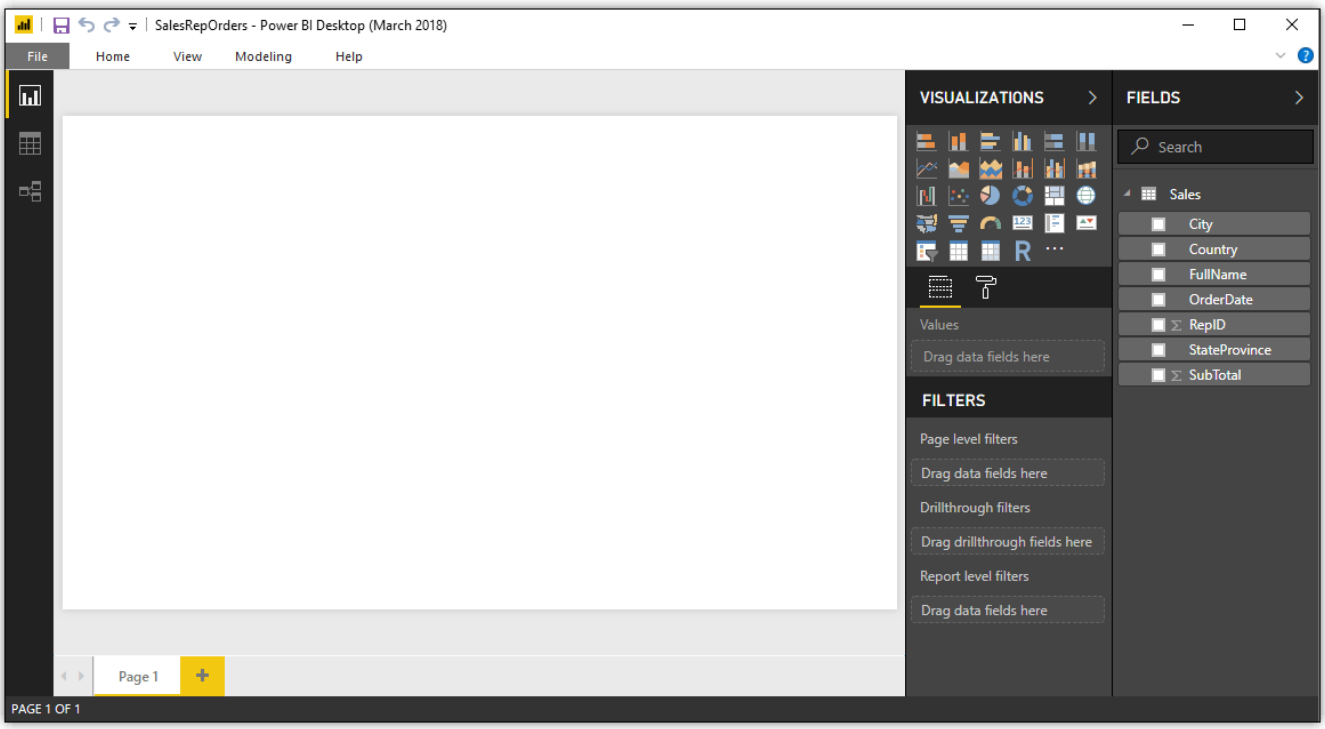
To update for the *OrderDate* column, select the dataset in *Data* view and then select the *OrderDate* column. On the *Modeling* ribbon, select *Date* from the *Date type* drop-down list. Next, click the *Format* drop-down list, point to *Date Time,* and click the format *2001-03-14 (yyyy-MM-dd)*.

To update the *SubTotal* column, select the column in *Data* view and then select *Whole Number* from the *Data type* drop-down list. You should receive a warning message about the loss of data precision. Click *Yes* to continue. In the *Currency format* drop-down list, click *$ English (United States)*. In the *Decimal places* box, set the number of decimal places to *0*.

Getting started with Power BI Desktop Reports

With the dataset in place, you're ready to start adding visualizations to the report. But before you do, save the file if you haven't already done so. The file provides the structure necessary for publishing the report—along with its visualizations and datasets—to the Power BI service. Each file is associated with a single report made up of one or more pages, with each page containing the actual visualizations.

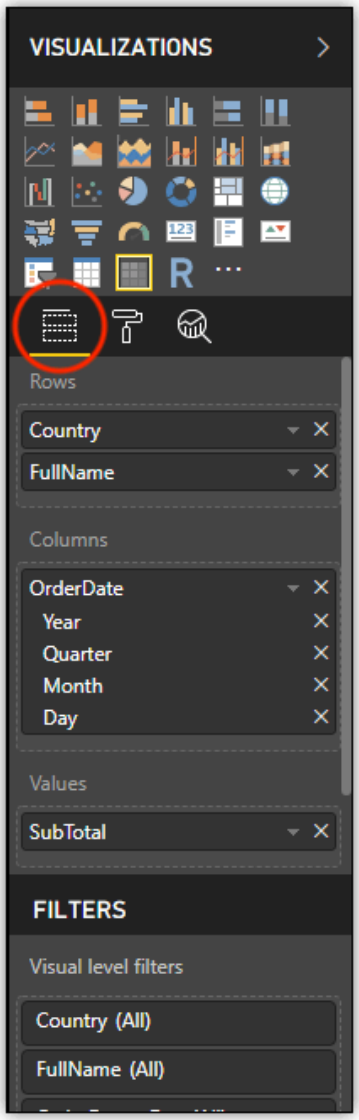
To add a visualization to a report, go to *Report* view, where you're presented with a design surface and the *Visualizations* and *Fields* panes, as shown in the following figure. The design surface contains the report pages, with *Page 1* already added to the report.



The *Visualizations* pane provides the tools necessary for adding and configuring the visualizations. The *Fields* pane includes a list of the defined datasets, with access to each dataset column. The above figure indicates that the *Sales* dataset is the only visible dataset included in this report file. (In Power BI Desktop, you can configure a dataset to be hidden in *Report* view.)

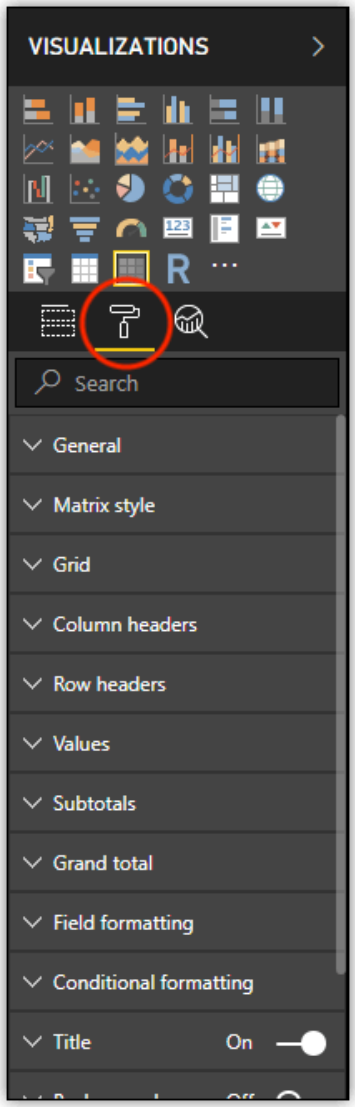
Most of the tasks you carry out in *Report* view are point-and-click or drag-and-drop operations. For example, to add a visualization to a report page, you simply click the visualization's button at the top of the *Visualizations* pane. Power BI Desktop will add the visualization to the design surface, where you can drag it to a different position or resize its window. You can then specify what data to add to the visualization by selecting columns from the *Fields* pane.

When a visualization is selected on the design surface, Power BI Desktop updates the *Visualizations* pane to include configuration options specific to that visualization. For example, the following figure shows the *Visualizations* pane with a *Matrix* visual selected on the design surface. In this case, the *Country* and *FullName* columns have been added to the *Rows* section and the *OrderDate* column has been added to the *Columns* section, with the date hierarchy expanded. (We'll be getting into the details of how to add columns in just a bit.)



The top of the *Visualizations* pane contains the icon buttons you use to add visualizations to a report. The rest of the pane is specific to configurating the selected visualization. This part of the pane is divided into three tabs: *Fields, Format,* and *Analytics*. In the above figure, the *Fields* tab is selected, as indicated by the red circle. The options on this tab are used to apply data to the visualization.

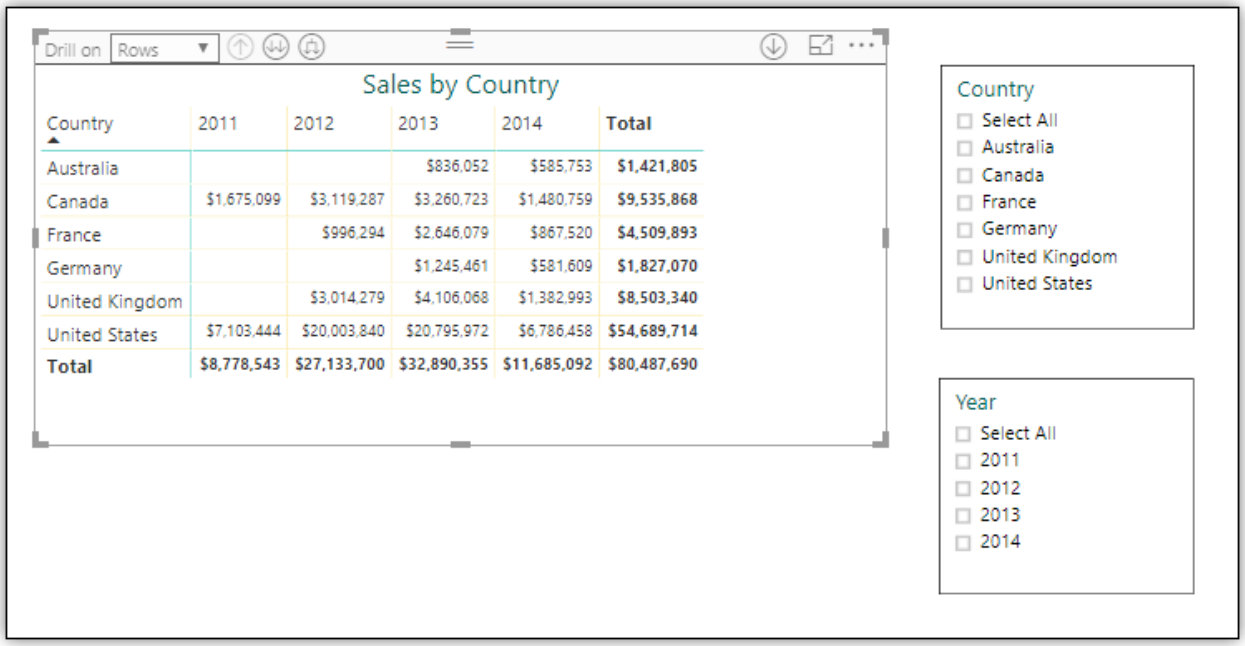
The second tab, *Format,* provides options for configuring how the selected visualization appears. The options are separated into categories, which are specific to the selected visualization. The following figure shows the *Format* categories for the *Matrix* visual.



The *Analytics* tab lets you add dynamic reference lines to certain types of visualizations. Later in the article, we'll be discussing this tab in more detail.

Creating a Matrix Visual

The best way to understand how to work with visualizations is to set them up for yourself. In this way, you get hands-on experience with the tools, while seeing how all the pieces fit together. For the first example, you'll add and configure a *Matrix* visual and two *Slicer* visuals. The following figure shows what they'll look like after you've set them up. The visualization on the left is the *Matrix* visual and the two on the right are the *Slicer* visuals. The *Matrix* visual is selected in the figure, as indicated by the menu across the top of the visualization and the frame that surrounds it.



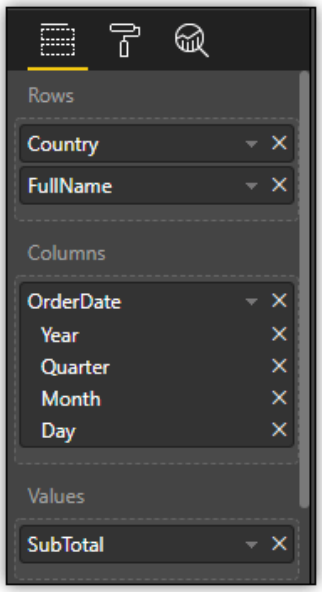
Country	2011	2012	2013	2014	Total
Australia			$836,052	$585,753	$1,421,805
Canada	$1,675,099	$3,119,287	$3,260,723	$1,480,759	$9,535,868
France		$996,294	$2,646,079	$867,520	$4,509,893
Germany			$1,245,461	$581,609	$1,827,070
United Kingdom		$3,014,279	$4,106,068	$1,382,993	$8,503,340
United States	$7,103,444	$20,003,840	$20,795,972	$6,786,458	$54,689,714
Total	**$8,778,543**	**$27,133,700**	**$32,890,355**	**$11,685,092**	**$80,487,690**

The *Matrix* visual aggregates measure data across both columns and rows, while supporting extensive drill-down capabilities, depending on the data and how you've configured the matrix.

To add the Matrix shown in the previous figure, take the following steps:

1. Click the *Matrix* button in the top section of the *Visualizations* pane. This adds a placeholder object to the design surface, which is already selected.
2. Drag the *Country* column from the *Fields* pane to the *Rows* section on the *Visualizations* pane.
3. Drag the *FullName* column from the *Fields* pane to the *Rows* section on the *Visualizations* pane, directly beneath the *Country* column. By adding the *FullName* column after the *Country* column, you make it possible for users to drill down into the country to view data about the sales reps in that country.
4. Drag the *OrderDate* column from the *Fields* pane to the *Columns* section on the *Visualizations* pane. Because the *OrderDate* column contains date values, Power BI Desktop automatically defines hierarchical levels that can be used to drill down into the data based on dates.
5. Drag the *SubTotal* column from the *Fields* pane to the *Values* section on the *Visualizations* pane. The *SubTotal* column provides the measure on which the aggregations are based.

The *Fields* tab on the *Visualizations* pane should now look like the one in the following figure.

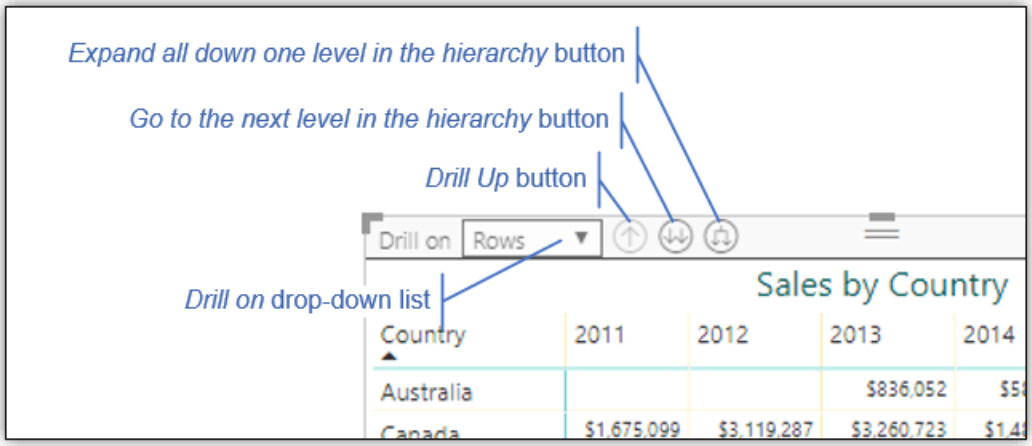


After you add the data you need, you can configure the visual's settings on the *Format* tab to ensure it looks just the way you want, and the data can be easily understood. To make your matrix look similar to the one shown in the above figure, take the following steps:

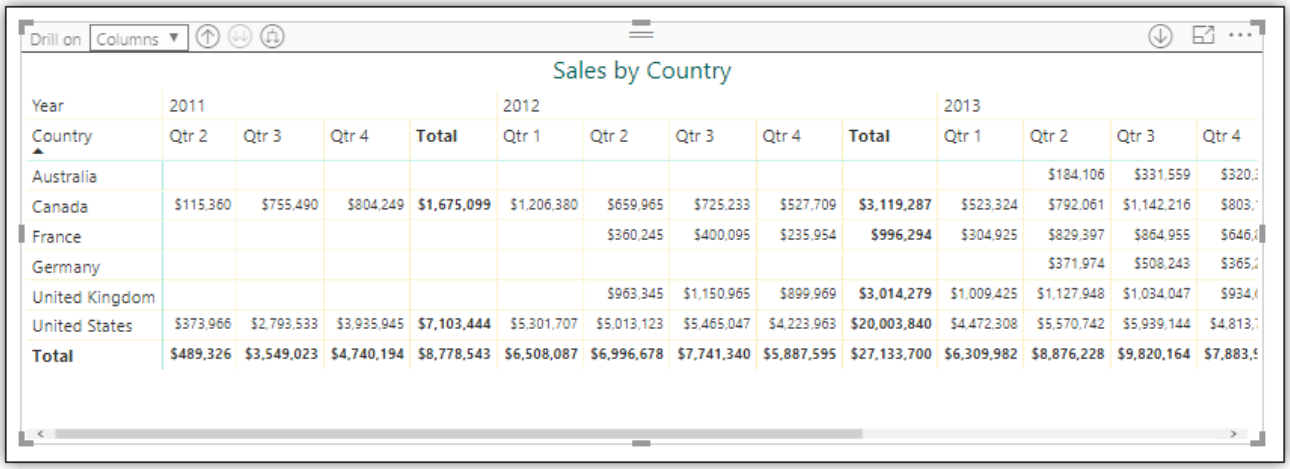
1. Expand the *Matrix style* section and select *Minimal* from the *Style* drop-down list.
2. Expand the *Grid* section and set the *Vert grid* option to *On,* and then select the lightest yellow color for the *Vert grid color* option.
3. Expand the *Column headers* section and set the *Text size* option to *12*.
4. Expand the *Row headers* section and set the *Text size* option to *12*.
5. Expand the *Values* section and set the *Text size* option to *10*.
6. Expand the *Subtotals* section and set the *Text size* option to *10*.
7. Expand the *Grand total* section and set the *Text size* option to *10*.
8. Expand the *Title* section and set the *Title* option to *On*. In the *Title Text* box, type *Sales by Country,* and set the *Font color* option to the darkest aquamarine. In the *Alignment* subsection, click the *Center* option, and then set the *Text size* option to *17*.
9. In the *Border* section, set the option to *On*.

Your *Matrix* visual should now be in fairly good shape. Going forward, I'll leave it up to you to decide how to configure the settings on the *Format* tab, based on your own preferences. The main thing to keep in mind when formatting a visualization is to make sure it's selected on the design surface.

When you select a *Matrix* visual on the design surface, drop-down list is available for choosing whether drill-down operations are based on columns or rows, as shown in the following figure.



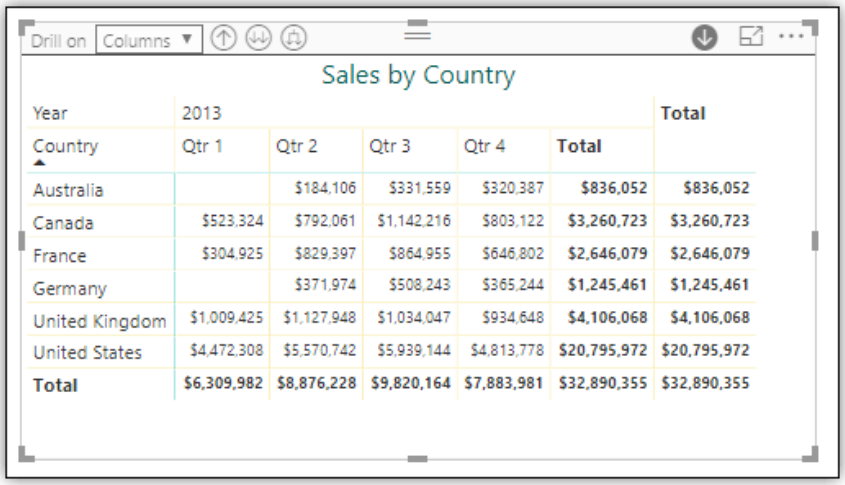
To test this out, select *Columns* from the *Drill on* drop-down list, and then click the button *Expand all down one level in the hierarchy* (third button to the right of the drop-down list). When you click the button, the matrix provides a breakdown of the aggregated data based on quarters as well as years, as shown in the following figure.



Year	2011				2012					2013			
Country	Qtr 2	Qtr 3	Qtr 4	**Total**	Qtr 1	Qtr 2	Qtr 3	Qtr 4	**Total**	Qtr 1	Qtr 2	Qtr 3	Qtr 4
Australia											$184,106	$331,559	$320,
Canada	$115,360	$755,490	$804,249	**$1,675,099**	$1,206,380	$659,965	$725,233	$527,709	**$3,119,287**	$523,324	$792,061	$1,142,216	$803,
France						$360,245	$400,095	$235,954	**$996,294**	$304,925	$829,397	$864,955	$646,
Germany											$371,974	$508,243	$365,
United Kingdom						$963,345	$1,150,965	$899,969	**$3,014,279**	$1,009,425	$1,127,948	$1,034,047	$934,
United States	$373,966	$2,793,533	$3,935,945	**$7,103,444**	$5,301,707	$5,013,123	$5,465,047	$4,223,963	**$20,003,840**	$4,472,308	$5,570,742	$5,939,144	$4,813,
Total	**$489,326**	**$3,549,023**	**$4,740,194**	**$8,778,543**	**$6,508,087**	**$6,996,678**	**$7,741,340**	**$5,887,595**	**$27,133,700**	**$6,309,982**	**$8,876,228**	**$9,820,164**	**$7,883,**

To return to the previous view, click the *Drill Up* button just to the right of the *Drill on* drop-down list. Next, click the button *Click to turn on Drill Down* near the top right corner to turn on drill-down capabilities directly within the matrix. When this button is selected, its shading is reversed.

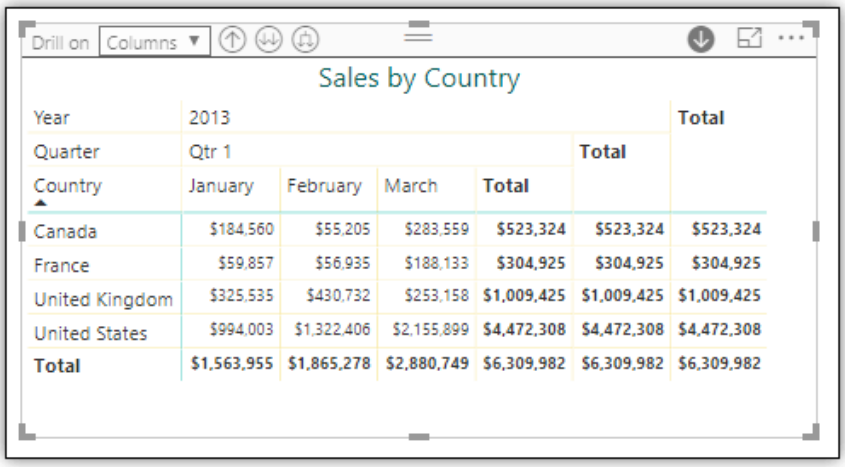
With drill-down enabled, click the *2013* column header. The matrix will now display data specific to that year, as shown in the following figure.



Sales by Country

Year	2013					Total
Country	Qtr 1	Qtr 2	Qtr 3	Qtr 4	Total	
Australia		$184,106	$331,559	$320,387	$836,052	$836,052
Canada	$523,324	$792,061	$1,142,216	$803,122	$3,260,723	$3,260,723
France	$304,925	$829,397	$864,955	$646,802	$2,646,079	$2,646,079
Germany		$371,974	$508,243	$365,244	$1,245,461	$1,245,461
United Kingdom	$1,009,425	$1,127,948	$1,034,047	$934,648	$4,106,068	$4,106,068
United States	$4,472,308	$5,570,742	$5,939,144	$4,813,778	$20,795,972	$20,795,972
Total	**$6,309,982**	**$8,876,228**	**$9,820,164**	**$7,883,981**	**$32,890,355**	**$32,890,355**

As already noted, Power BI Desktop automatically creates a hierarchy based on the date values, with years at the top, then quarters, next months, and finally days. You can drill down into all levels within a *Matrix* visual. For example, to drill into the first quarter, click the *Qtr 1* column header, which allows you to view sales data at the month level, as shown in the following figure.



Sales by Country

Year	2013					Total
Quarter	Qtr 1				Total	
Country	January	February	March	Total		
Canada	$184,560	$55,205	$283,559	$523,324	$523,324	$523,324
France	$59,857	$56,935	$188,133	$304,925	$304,925	$304,925
United Kingdom	$325,535	$430,732	$253,158	$1,009,425	$1,009,425	$1,009,425
United States	$994,003	$1,322,406	$2,155,899	$4,472,308	$4,472,308	$4,472,308
Total	**$1,563,955**	**$1,865,278**	**$2,880,749**	**$6,309,982**	**$6,309,982**	**$6,309,982**

If you select the *Rows* option in the *Drill on* drop-down list, instead of the *Columns* option, you will see different results when drilling into the data. For example, if you click the name of a country in the *Country* column, the matrix will display a list of sales reps for that country, along with their sales totals. Give it a try and see how to goes.

You can also view a visualization in Focus mode, which enlarges the visualization so that it uses the entire design surface. To view a visual in focus mode, click the *Focus mode* button near the top right corner. Your *Matrix* visual should now look similar to the one in the following figure. To return back to the regular view of the design surface, click the *Back to Report* button.

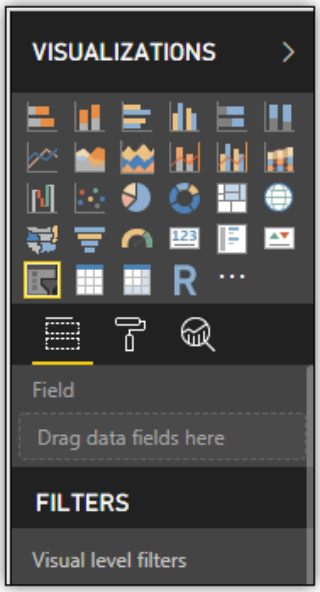
Drill on Columns

Back to Report | SALES BY COUNTRY

Country	2011	2012	2013	2014	Total
Australia			$836,052	$585,753	**$1,421,805**
Canada	$1,675,099	$3,119,287	$3,260,723	$1,480,759	**$9,535,868**
France		$996,294	$2,646,079	$867,520	**$4,509,893**
Germany			$1,245,461	$581,609	**$1,827,070**
United Kingdom		$3,014,279	$4,106,068	$1,382,993	**$8,503,340**
United States	$7,103,444	$20,003,840	$20,795,972	$6,786,458	**$54,689,714**
Total	**$8,778,543**	**$27,133,700**	**$32,890,355**	**$11,685,092**	**$80,487,690**

You can also add slicers to your reports that provide a way for users to filter the data within the visualization. For this exercise, you will add two slicers, one for countries and one for years.

To add the *Country* slicer, verify that no visualizations are selected on the design surface, and then click the *Slicer* button at the top of the *Visualizations* pane, as shown in the following figure. The *Slicer* button is outlined in yellow to indicate that it is the selected visual type.

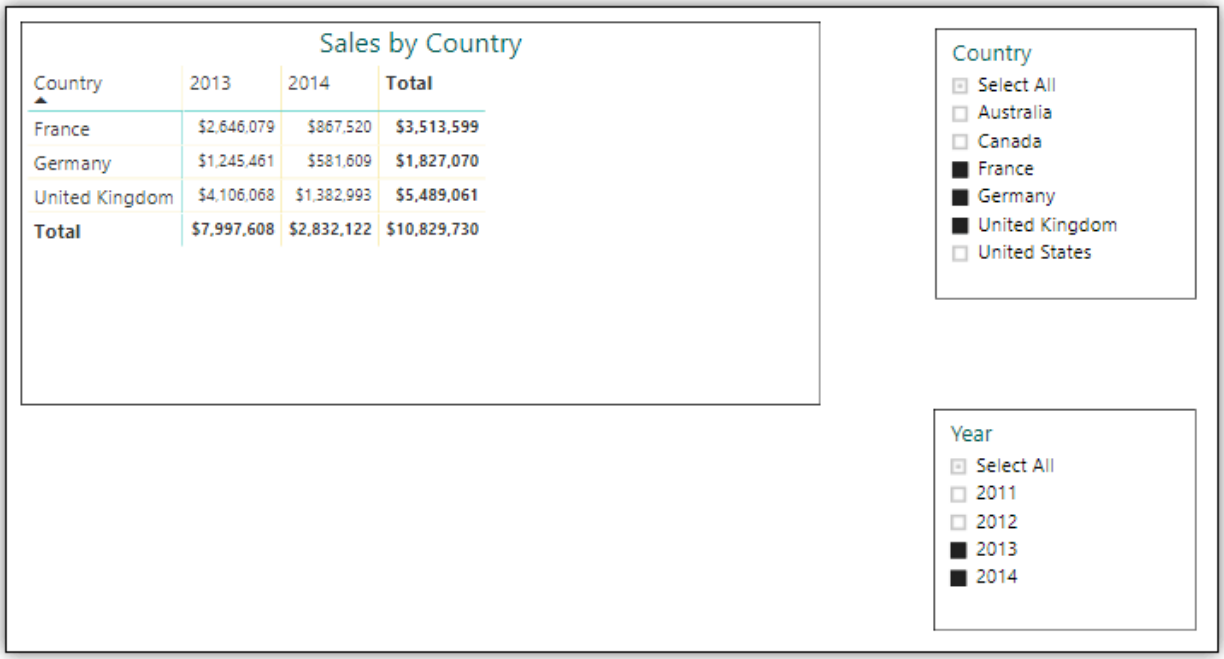


Next, drag the *Country* column from the *Fields* pane to the *Field* section in the *Visualizations* pane. You can then move and resize the slicer, as well as apply formatting. Power BI Desktop will automatically link the slicer to any related visualizations on that page. You might also need to change the slicer type from a dropdown list to a regular list of items so the slicer is more readable. To change the type, click the down arrow within the slicer visual itself, near the top right corner, and then select *List*.

To add the *Year* slicer, take the same steps as you did for the *Country* slicer, only this time, add the *OrderDate* column. Next, click the column's down arrow (on the *Visualizations* pane), and click *Date Hierarchy*. Power BI Desktop adds only the *Year* level to the slicer. Again, you might need to change the slicer type from *Dropdown* to *List*.

When formatting the two slicers, you can add a *Select All* value to the list of available values. To add the value, go to the *Format* tab on the *Visualizations* pane, expand the *Selection Controls* category, and set the *Show "Select All"* option to *On*.

To test the new slicers, select *France, Germany,* and *United Kingdom* in the *Country* slicer, and select *2013* and *2014* in the *Year* slicer. To select multiple values in a slicer, select the first value, press *Control*, and select the remaining values. Your visualizations should now look like those shown in the following figure.



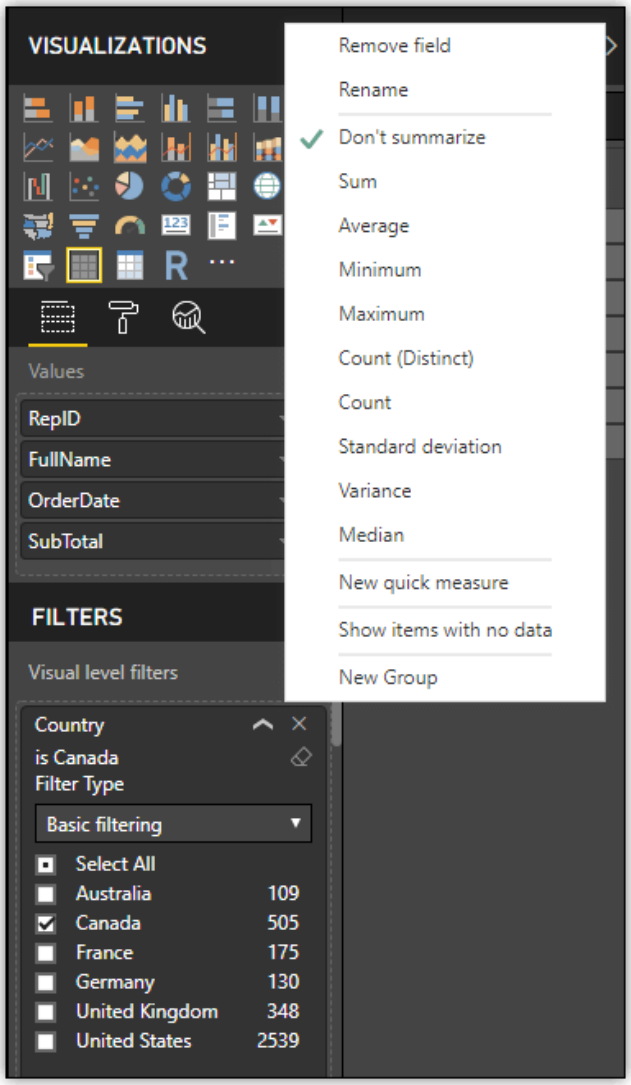
Once you've mastered how to add and configure a *Matrix* visual, you'll find that many of the concepts introduced here will apply to other visualizations. Just be certain to save your report file regularly to ensure that none of your changes get lost.

Creating Table and Card Visuals

When you add columns to a visualization, Power BI Desktop will often summarize the data to provide a big-picture perspective of the information. You saw this with the *Matrix* visual in the previous section, in which the *SubTotal* column (the measure) was aggregated across the *Country* and *OrderDate* columns to provide subtotals for each discrete group.

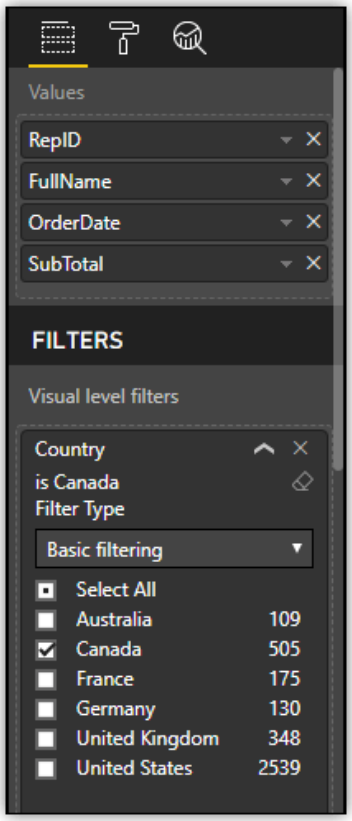
In some cases, you might not want the data automatically aggregated. For example, you might want to add a *Table* visual that lists individual Canadian sales only. To add this table, click the *Plus* button at the bottom of the design surface to insert a second page. Next, click the *Table* button on the *Visualizations* pane, and then add the *RepID, FullName, OrderDate,* and *SubTotal* columns to the *Values* section of the *Visualizations* tab.

Power BI Desktop will automatically try to summarize the *RepID* and *SubTotal* columns because they contain numerical values, using the *Count* and *Sum* aggregate functions, respectively. For each of these two columns, click the column's down arrow (in the *Visualizations* pane), and select *Don't summarize,* as shown in the following figure. Power BI Desktop will update the data to include individual rows.

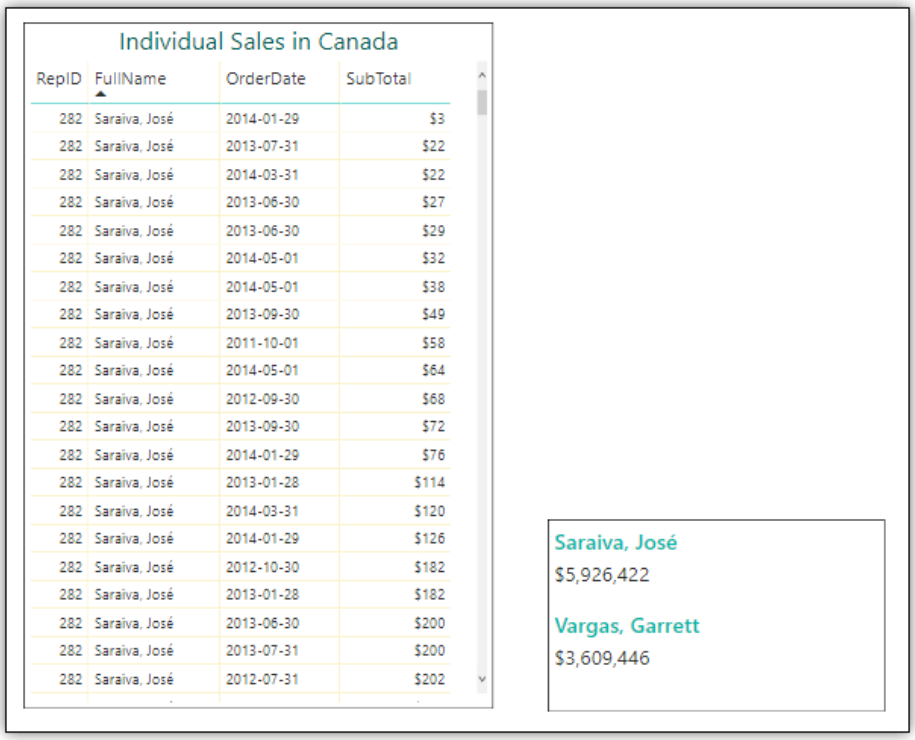


When you add a date column to the *Table* visual, Power BI Desktop automatically breaks the dates into hierarchical levels, which results in four columns (*Year, Quarter, Month,* and *Day*). If you want a single date column instead, you must reset the *OrderDate* column. In the *Visualizations* pane, click the column's down-arrow and select *OrderDate*. Power BI Desktop will update the table so the entire *OrderDate* value is listed in a single column.

The last step is to filter out all rows except the Canadian sales. To filter the data, drag the *Country* column from the *Fields* pane to the *Visual level filters* subsection in the *Filters* section of the *Visualizations* pane. Expand the *Country* section if not already expanded and select the *Canada* checkbox, as shown in the following figure.



Once you've set up the data, position and resize the *Table* visual and apply the necessary formatting. Your table should now look similar to the one shown in the following figure but formatted according to your preferences.



The figure also shows a second visual, which is a *Multi-row card.* In this case, the card displays the name of each Canadian sales rep, along with the total sales for that rep.

To add this card, verify that the *Table* visual is not selected and then click the *Multi-row card* button on the *Visualizations* pane. Next, drag the *FullName* column from the *Fields* pane to the *Fields* section of the *Visualizations* pane, and then do the same for the *SubTotal* column. Finally, set up the same filter you set up for the *Table* visual so that only Canadian sales reps are listed. Position, resize, and format the card as necessary, and be sure to save the report file.

Creating Pie, Donut, and Treemap Charts

In the next example, you'll add a page to the report and then add three visuals: *Pie chart, Donut chart,* and *Treemap*. The visualizations will display sales information for all countries except the United States. Although you'll be creating three different types of visualizations, you'll configure them exactly the same.

To add the *Pie chart* visual, take the following steps:

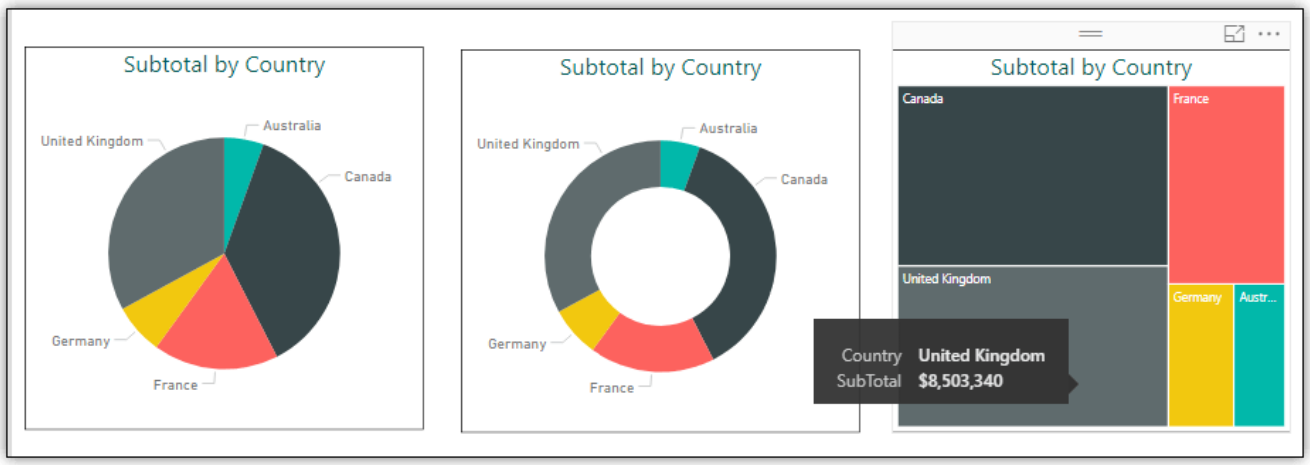
1. Click the *Plus* button to add a new page, and then click the *Pie chart* button on the *Visualizations* pane.
2. Drag the *Country* column from the *Fields* pane to the *Details* section of the *Visualizations* pane.
3. Drag the *Subtotal* column from the *Fields* pane to the *Values* section of the *Visualizations* pane.
4. Expand the *Country* column in the *Filters* section and select all countries except the US.
5. Position, resize, and format the visualization as necessary.

Once you have the first visualization exactly as you want to, you can create the next one, using the following steps:

1. Ensure that the *Pie chart* visual is selected.
2. Click *Copy* on the *Home* ribbon, and then click *Paste*. A copy of the original visualization is added to the design surface.
3. Drag the copied visual to its new position.
4. Click the *Donut chart* button on the *Visualizations* pane.

When you click one of the visualization buttons and a visualization is already selected, Power BI Desktop tries to apply the newly selected visual to the original configuration. This means that the visuals need to compatible with each other for this to work, like the two you just added.

The next step is to add the *Treemap* visual, taking the same approach as with the *Donut chart* visual. Copy and paste one of the other visuals and then click the *Treemap* button at the top of the *Visualizations* pane. When you're finished, you should end up with three visuals similar to those shown in the following figure.



The visuals are positioned in the order they were created: *Pie chart, Donut chart,* and *Treemap*. Normally, you would not add such similar visuals to the same page unless you had a compelling reason, but what we've done here demonstrates how easy it is to copy a visualization and change it to a different type.

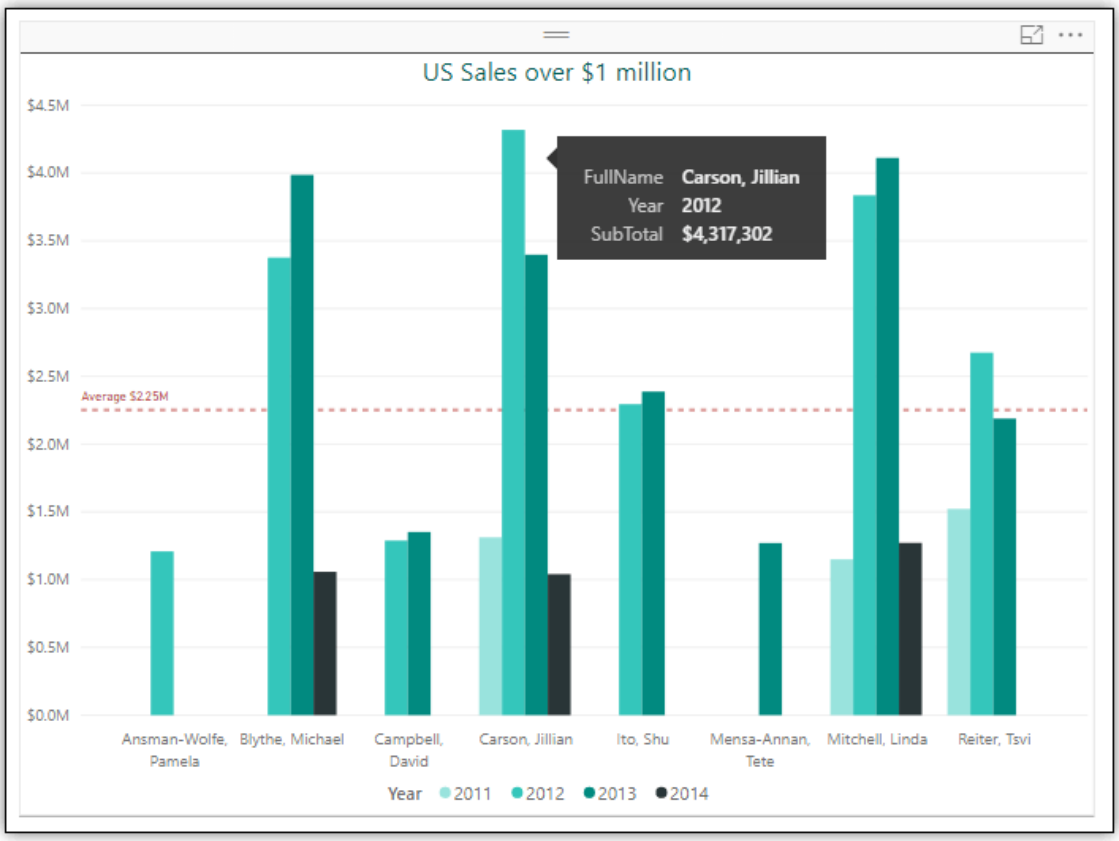
In the figure above, the *Treemap* visual also shows a pop-up message, which appears when you hover over a section of the visual. In this case, the pop-up is showing the total sales for the United Kingdom, but you can get information for any category on any of the three visualizations. In fact, most visualizations in Power BI Desktop support this feature.

Creating a Clustered Column Chart

The next visualization you will add is a *Clustered column chart* that shows US sales information for each sales rep with over $1 million in annual sales. The following steps walk you through the process of adding the visualization:

1. Click the *Plus* button to add a new page to the report, and then click the *Clustered column chart* button on the *Visualizations* pane.
2. Drag the *FullName* column from the *Fields* pane to the *Axis* section of the *Visualizations* pane. The axis provides the foundation on which the chart is built.
3. Drag the *OrderDate* column from the *Fields* pane to the *Legend* section of the *Visualizations* pane. The legend determines how data will be clustered.
4. In the *Visualizations* pane, click the *OrderDate* column's down arrow, and then select *Date Hierarchy*. Power BI Desktop automatically limits the hierarchy to the *Year* level.
5. Drag the *SubTotal* column from the *Fields* pane to the *Value* section of the *Visualizations* pane. The value serves as the measure used for aggregating the data.
6. Drag the *Country* column from the *Fields* pane to the *Visual level filters* subsection in the *Filters* section of the *Visualizations* pane. Expand the *Country* section and select *United States* to include only that country.
7. In the *Filters* section of the *Visualizations* pane, expand the *SubTotal* section, select *is greater than* from the first drop-down list, type *1000000* in the text box directly below the list, and click *Apply filter*. By applying this filter, only sales totals greater than $1 million will be included in the visualization.
8. Position, resize, and format the visualization as necessary.

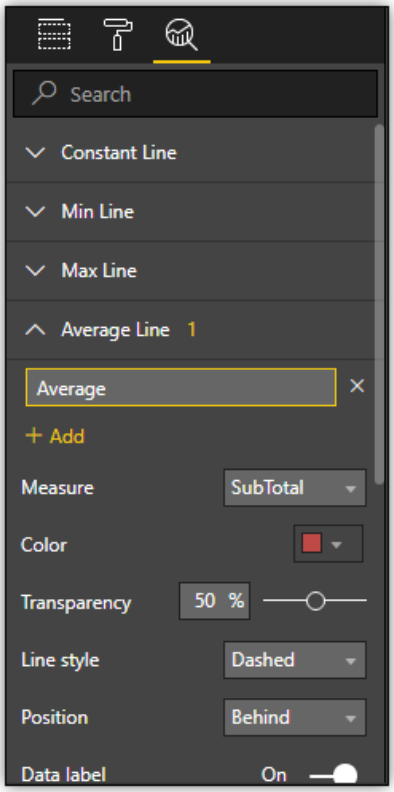
Your visualization should now look similar to the one shown in the following figure, except that it won't include the red analytics line. You'll be adding that shortly.



Analytics is a feature in Power BI Desktop that lets you add dynamic reference lines to a visual to provide additional insights into the data or reflect important trends. The analytic in the above figure shows the average sales amount based on values in the *SubTotal* column. To add this analytic, take the following steps:

1. Ensure that the *Clustered column chart* visual is still selected on the design surface.
2. In the *Visualizations* pane, go to the *Analytics* tab, expand the *Average Line* section, and click *Add.*
3. In the text box at the top of the *Average Line* section, change the default name to *Average*.
4. From the *Measure* drop-down list, select *SubTotal* if it's not already selected.
5. Configure the formatting settings as desired. You format the analytic directly on the *Analytics* tab when adding the analytic. For example, you can configure the analytic's color, percentage of transparency, and line style.

The following figure shows how I configured the analytic on my system.



You can add whatever analytics you think will be serve your purpose. Note, however, that not all visualizations support analytics.

Creating a Filled Map

In this last example, you'll add a *Filled map* visual and then a *Table* visual that contains data corresponding to the map. The *Filled map* visual uses patterns or shading to show how values differ across geographic regions. The visual relies on Microsoft's Bing service to provide default map longitude and latitude coordinates. As a result, you must be online when working with the *Filled map* visual for the map to fully render on the design surface.

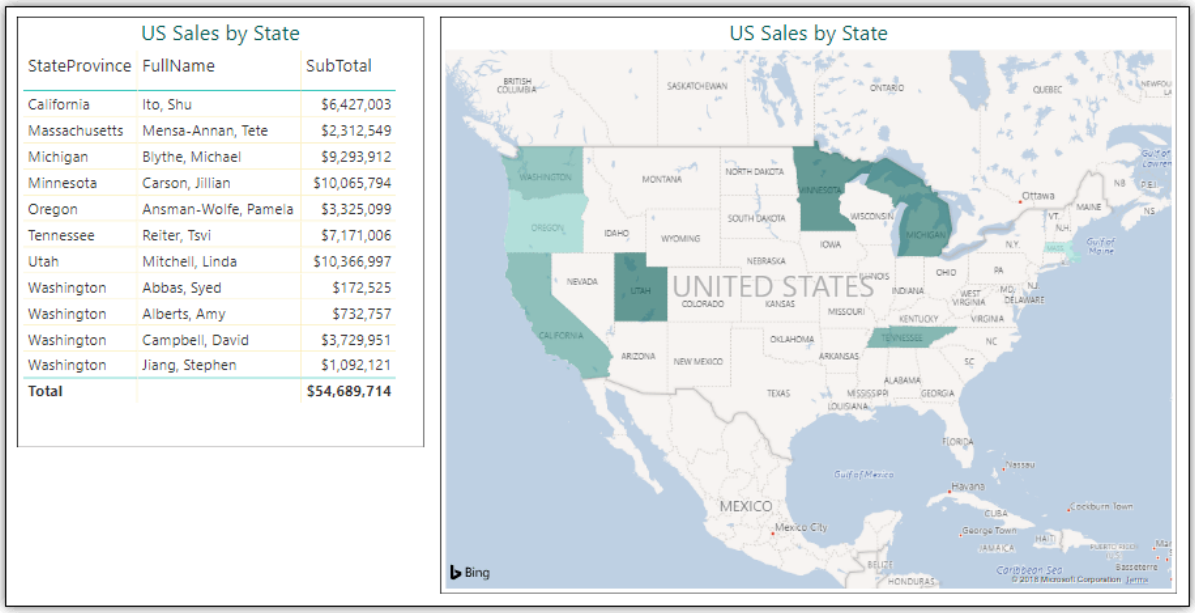
To add a *Filled map* visual based on the *Sales* dataset, take the following steps:

1. Click the *Plus* button to add a new page to the report, and then click the *Filled map* button on the *Visualizations* pane.
2. Drag the *StateProvince* column from the *Fields* pane to the *Location* section of the *Visualizations* pane. Power BI Desktop will automatically use these values to come up with their coordinates, leveraging the Bing service.
3. Drag the *SubTotal* column from the *Fields* pane to the *Color saturation* section of the *Visualizations* pane. The column serves as the measure used for aggregating totals and arriving at the state shading.
4. Drag the *Country* column from the *Fields* pane to the *Visual level filters* subsection in the *Filters* section of the *Visualizations* pane. Expand the *Country* section and select *United States.*.
5. Position, resize, and format the visualization as necessary.

That's all there is to adding the *Filled map* visual. You can then add the *Table* visual, using the following steps:

1. Ensure that the *Filled map* visual is not selected on the design surface, and then click the *Table* button on the *Visualizations* pane.
2. Drag the *StateProvince, FullName,* and *SubTotal* columns from the *Fields* pane to the *Values* section of the *Visualizations* pane.
3. Drag the *Country* column from the *Fields* pane to the *Visual level filters* subsection in the *Filters* section of the *Visualizations* pane. Expand the *Country* section and select *United States.*
4. Position, resize, and format the visualization as necessary.

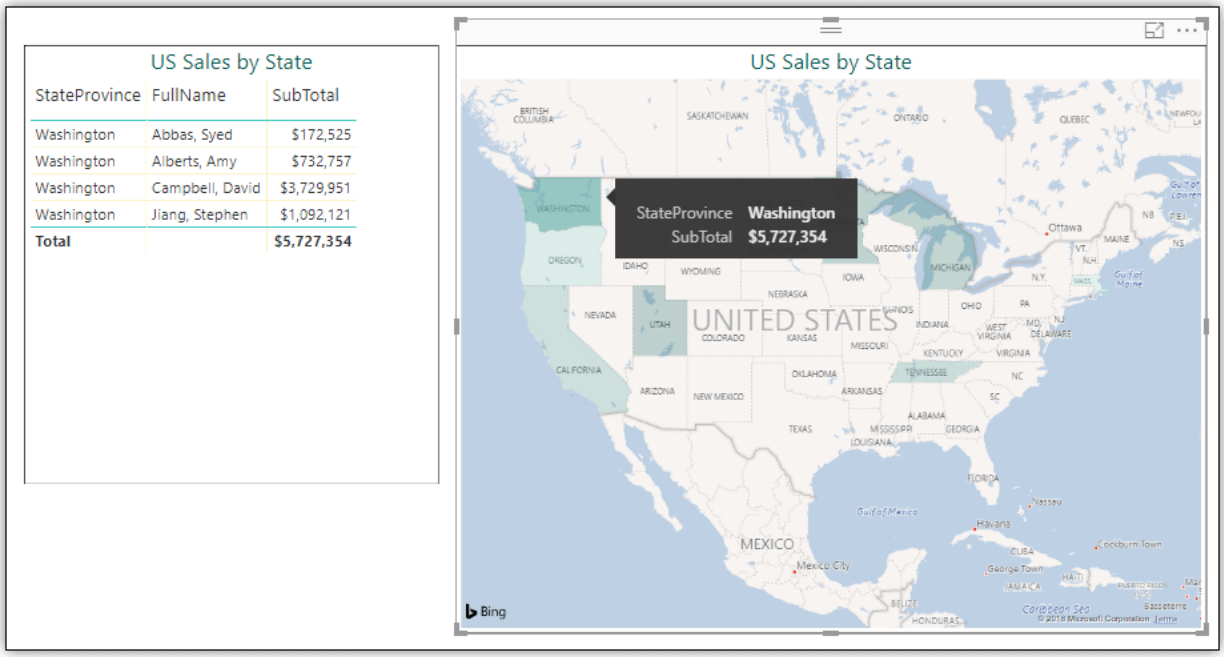
Your visualizations should now look similar to those shown in the following figure.



US Sales by State

StateProvince	FullName	SubTotal
California	Ito, Shu	$6,427,003
Massachusetts	Mensa-Annan, Tete	$2,312,549
Michigan	Blythe, Michael	$9,293,912
Minnesota	Carson, Jillian	$10,065,794
Oregon	Ansman-Wolfe, Pamela	$3,325,099
Tennessee	Reiter, Tsvi	$7,171,006
Utah	Mitchell, Linda	$10,366,997
Washington	Abbas, Syed	$172,525
Washington	Alberts, Amy	$732,757
Washington	Campbell, David	$3,729,951
Washington	Jiang, Stephen	$1,092,121
Total		**$54,689,714**

Power BI Desktop automatically links visualizations located on the same report page. As a result, when you select an item in one visualization, the data is updated in the other visualizations. For example, in the following figure, Washington state is selected in the *Filled map* visual. As a result, the *Table* visual now includes only data about sales in Washington state.



US Sales by State

StateProvince	FullName	SubTotal
Washington	Abbas, Syed	$172,525
Washington	Alberts, Amy	$732,757
Washington	Campbell, David	$3,729,951
Washington	Jiang, Stephen	$1,092,121
Total		**$5,727,354**

Linked visuals can be a handy way to allow users to gain quick insights into the same data from different perspectives, without having to create lots of extra visualizations. This also works for slicers as well, which are added at the page level, again providing a simple way to offer different perspectives into the data.

Lots More Where That Came From

Power BI Desktop supports a number of visualizations in addition to what we've covered here. For most of those visualizations, you can apply many of the same concepts that were demonstrated in this article. Power BI Desktop also supports the ability to use R scripts to create visualizations or to import R-based custom visualizations, as described in the Part 3 of this series.

Regardless of the visualization type, the best way to learn how to build reports in Power BI Desktop is to experiment with the visualizations themselves, trying out different ones against the same data or different data. The more time you spend working

directly with these features, the better you'll understand their capabilities and the quicker you'll be able to create visualizations that provide meaningful insights into the data. Power BI Desktop offers a solid foundation for building effective business intelligence reports, but it's up to you to figure out what it takes to get there.

Power BI Introduction: Publishing Reports to the Power BI Service — Part 8

Power BI Desktop is typically used to create reports that will eventually be published to Power BI Services where dashboards can be built. Robert Sheldon demonstrates how to publish reports from Power BI Desktop to the Power BI Service. Once published, the visualizations can be filtered, drilled into, or pinned to a dashboard.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Power BI Desktop plays a pivotal role in the Power BI suite of tools, which is why it's been the focus of much of this series. If you've been following along, you learned how to import and transform data, build reports and visualizations, incorporate R scripts, navigate the Power Query M formula language, and perform a variety of other tasks.

Although there is still much more you can do in Power BI Desktop, the ultimate goal is always the same—to build visually rich reports that provide stakeholders with actionable insights into the underlying data.

But building reports is only part of the equation. You must also make these reports available to the people who depend on the information they provide. For this, you need to publish the reports to the Power BI service or save them to Power BI Report Server (if you have a Power BI Premium subscription). Although you can share Power BI Desktop *.pbix* files with other users, this is not nearly as effective as disseminating the reports through one of the appropriate channels.

In this article, you'll learn how to publish a report to the Power BI service and work with the published report through the Power BI interface. The article steps you through such tasks as viewing and updating the report, saving report components to a dashboard, and creating a new report based on the published dataset. You'll also learn how you can update the report in Power BI Desktop and then republish it to the Power BI service.

Publishing to the Power BI Service

The examples in this article are based on the report file created in the previous article in this series, Building Reports in Power BI Desktop — Part 7. On my system, I named the report *SalesRepOrders,* which is the name used in the examples to follow. Figure 1 shows the report's *Matrix visual* page as it appears in *Report* view. The report is made up of five pages, with visuals on each page.

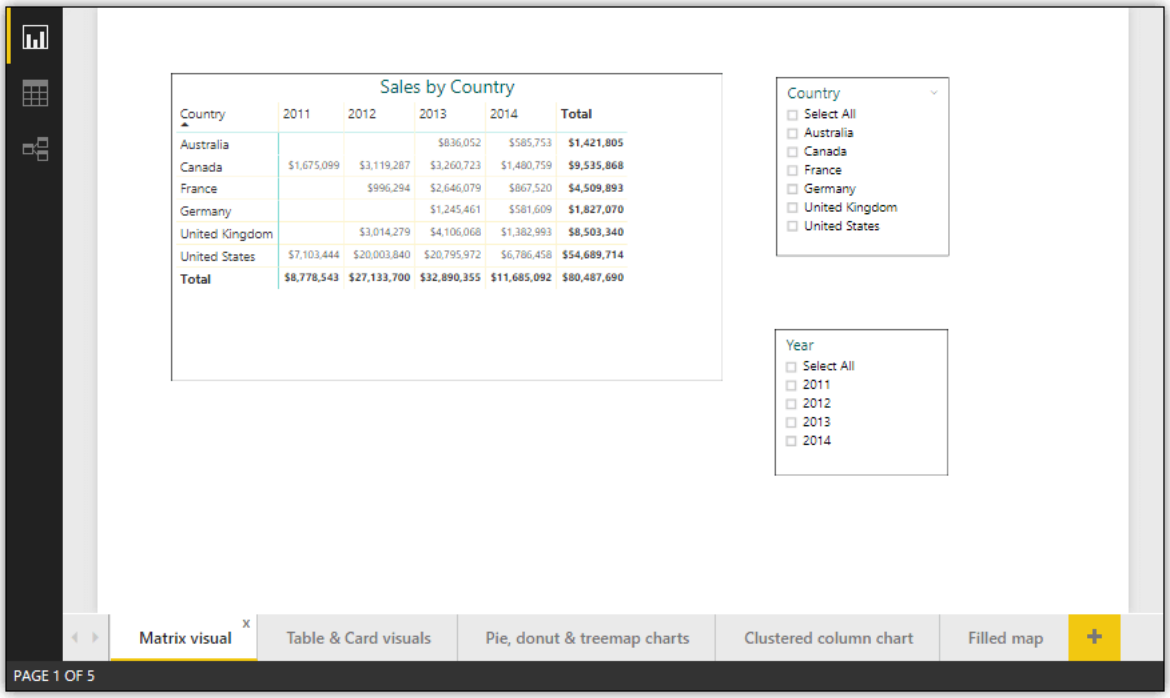


Figure 1. Viewing the *SalesRepOrders* report in Power BI Desktop

NOTE: To make it easier to follow along with this article, rename your visuals to match Figure 1 if you didn't name them when creating the report in the previous article.

To publish the *SalesRepOrders* report to Power BI, go to the *File* tab, point to *Publish,* and click *Publish to Power BI*. If you're not already signed into the Power BI service, you'll be prompted to provide your login credentials. Once you're connected, you'll then be prompted to choose a destination. Select the *My workspace* option if it is not selected, and then click *Select,* as shown in Figure 2.

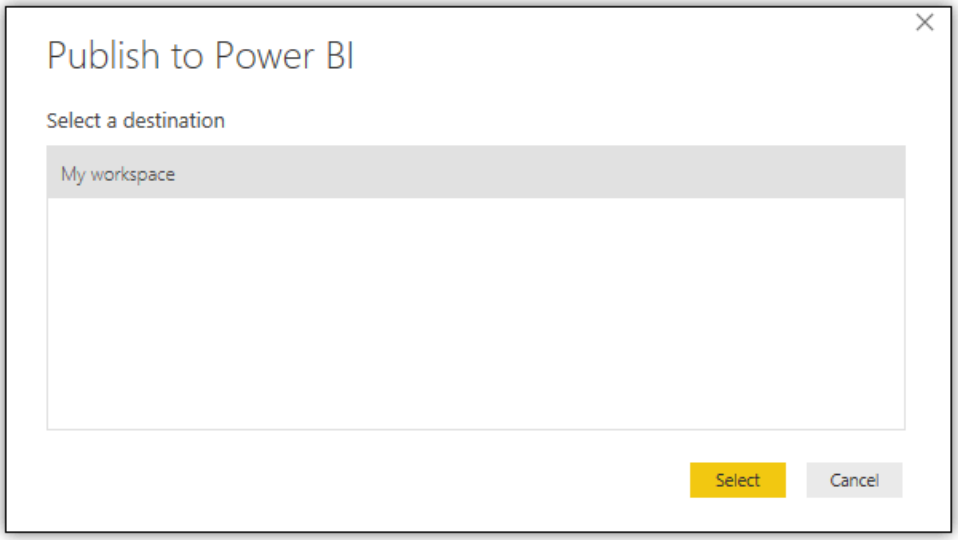


Figure 2. Publishing the *SalesRepOrders* report to *My workspace*

After you select a destination, the *Publishing to Power BI* dialog box will appear showing the publishing status. Ultimately, you should receive a *Success!* message, preceded by a green checkmark, as shown in Figure 3. The dialog box also displays a *Do You Know?* message that provides a tip about using the Power BI service.

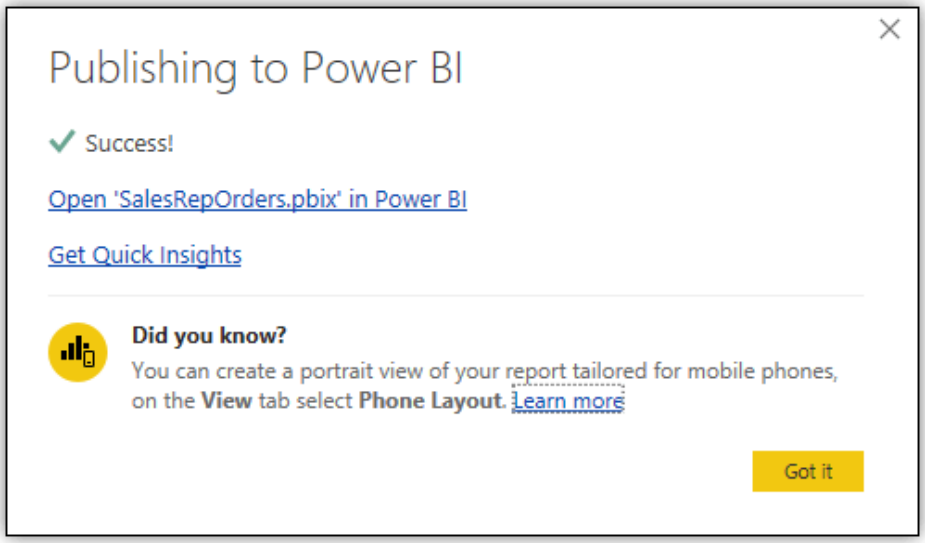


Figure 3. Verifying publication of the *SalesRepOrders* report

If you want to access the published report immediately, click the link *Open 'SalesRepOrders.pbix' in Power BI*. This will launch the Power BI service in your system's default browser. You might again be prompted for login credentials.

If you want to go to the Power BI site and automatically generate *insights* in the process, click the second link, *Get Quick Insights*. Insights are interactive visualizations that Power BI generates on demand. I've had mixed success when using this link.

If you don't want to connect to the Power BI service at this time, click the *Got it* button to close the dialog box.

Working with Reports in Power BI

You can access your published report through the Power BI site from any supported browser. For example, I've been able to access the site from Chrome and Edge in Windows and from Chrome and Safari in macOS. When you sign in, you're taken to a development interface that provides the tools necessary to import and visualize data. Figure 4 shows the interface after publishing the *SalesRepOrders* report file.

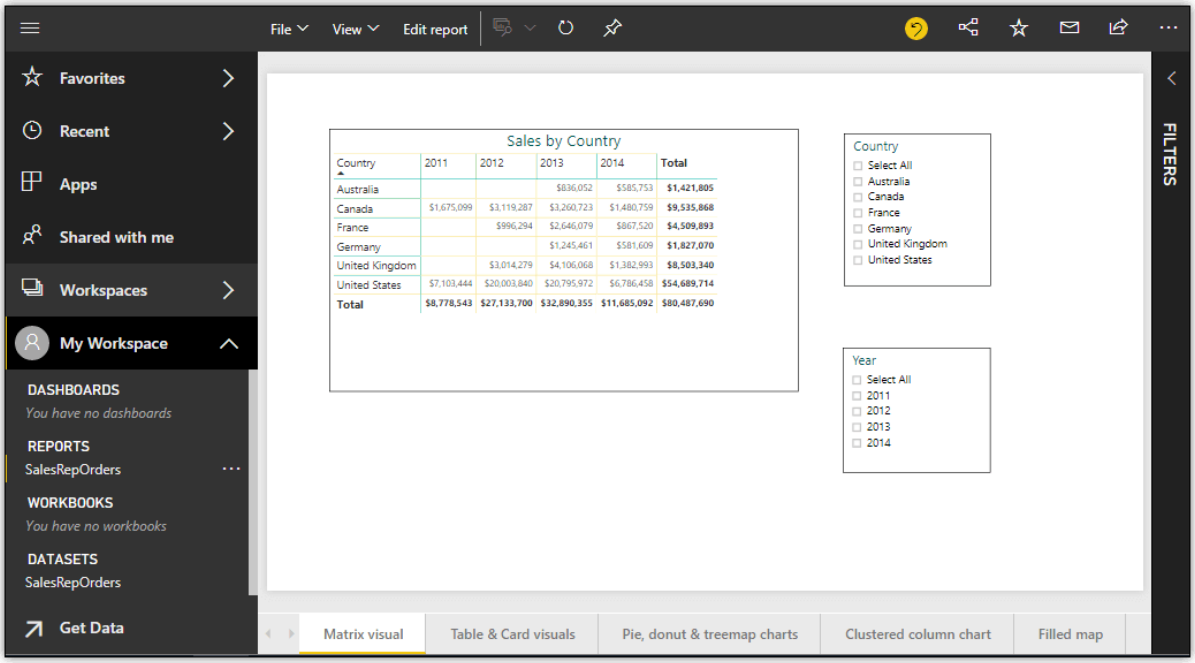


Figure 4. Viewing the *SalesRepOrders* report in the Power BI service

The *SalesRepOrders* report and dataset are listed in the *My Workspace* section in the left pane. (You might have to expand this section after signing into the service.) The *My Workspace* section is your personal work area for accessing and modifying your own dashboards, reports, and datasets.

From the *My Workspace* section, you can access resources in any of the following four categories:

- **Dashboards:** Canvases for presenting data through tiles or widgets. A dashboard can be associated with only one workspace, but it can display visualizations from multiple datasets or reports. If you're a Power BI Pro or Premium subscriber, you can also share dashboards.
- **Reports:** Collections of visualizations based on data in the defined datasets. A report can be associated with only one workspace, but it can be associated with multiple dashboards within that workspace. You can interact with a report either in *Reading view* or *Editing view,* depending on your granted level of permissions. Each report is made up of one or more pages.
- **Workbooks:** Special types of datasets created by *uploading* Microsoft Excel files to the Power BI service. You can upload an Excel file from within the Power BI service or by publishing the file directly from Excel. The workbook data requires no special formatting. This is different from *importing* an Excel file, which adds the dataset to the *Datasets* category. To import an Excel file, the data within the file must be formatted as an Excel table.
- **Datasets:** Collections of related data that you import or connect to. A dataset is similar to a database table and can be used in multiple reports, dashboards, and workspaces. You can retrieve data from files, databases, online services, or Power BI apps published by other people in your organization.

In Figure 4, the *SalesRepOrders* report in the *Reports* section is selected, with the report contents displayed in the main window. In this case, the *Matrix visual* page is selected, but you can choose any of the other pages to view those visuals, just like you saw in Power BI Desktop.

From the main *Reports* window, you can carry out several operations, such as saving a copy of the report, refreshing the data, sharing the report, or pinning the page to a dashboard which is covered later in the article.

By default, Power BI displays the report in *Reading view*. If you have the proper permissions, you can also work with the report in *Editing view.* To get into *Editing view,* click the *Edit report* button at the top of the window. This allows you to modify the report and visualizations directly within the Power BI interface.

Updating a visualization in the Power BI service is much like updating a visualization in Power BI Desktop, as shown in Figure 5. Notice that *Editing view* includes both the *Visualizations* pane and *Fields* pane.

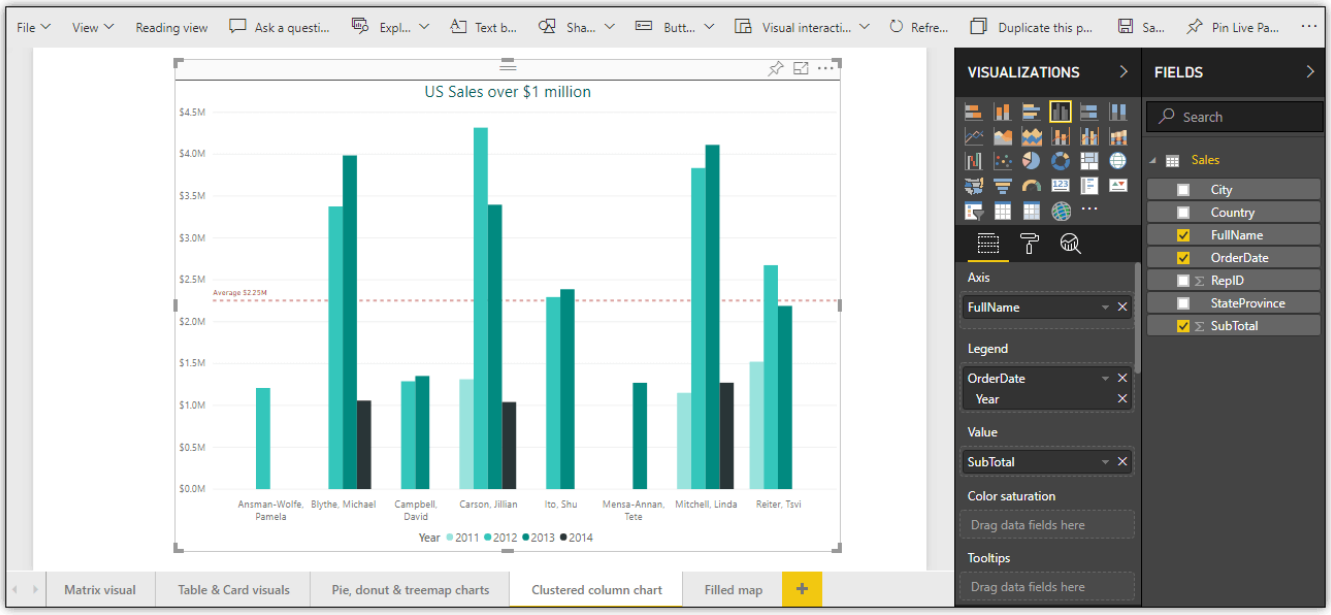


Figure 5. Editing the *SalesRepOrders* report in the Power BI service

A simple way to test the editing features is to update the report's *Clustered column chart* visual to display sales greater than $2 million, rather than $1 million. To update the visual, first make sure you have selected it and then expand the *SubTotal* column in the *Filters* section of the *Visualizations* pane, change the *1000000* value to *2000000,* and click *Apply filter*.

You can then save the report under the same name or save it to a different name using the options available to the *File* menu at the top of the design surface. (On my system, I saved the report to a different name so the original report remained unchanged.)

In addition to updating the visualizations, you can also modify the report itself. For example, you can insert a shape, button, text box, or *Ask a question* section. You'll find the tools for making these changes at the top of the design surface.

Updating and Republishing Reports

Rather than updating reports directly within the Power BI service, you might prefer to modify them in Power BI Desktop and then republish them to the Power BI service. For example, suppose you want update the *SalesRepOrders* report by adding drillthrough filters to the *Matrix* visual, using the *Country* and *FullName* columns for the filters, as shown in Figure 6.

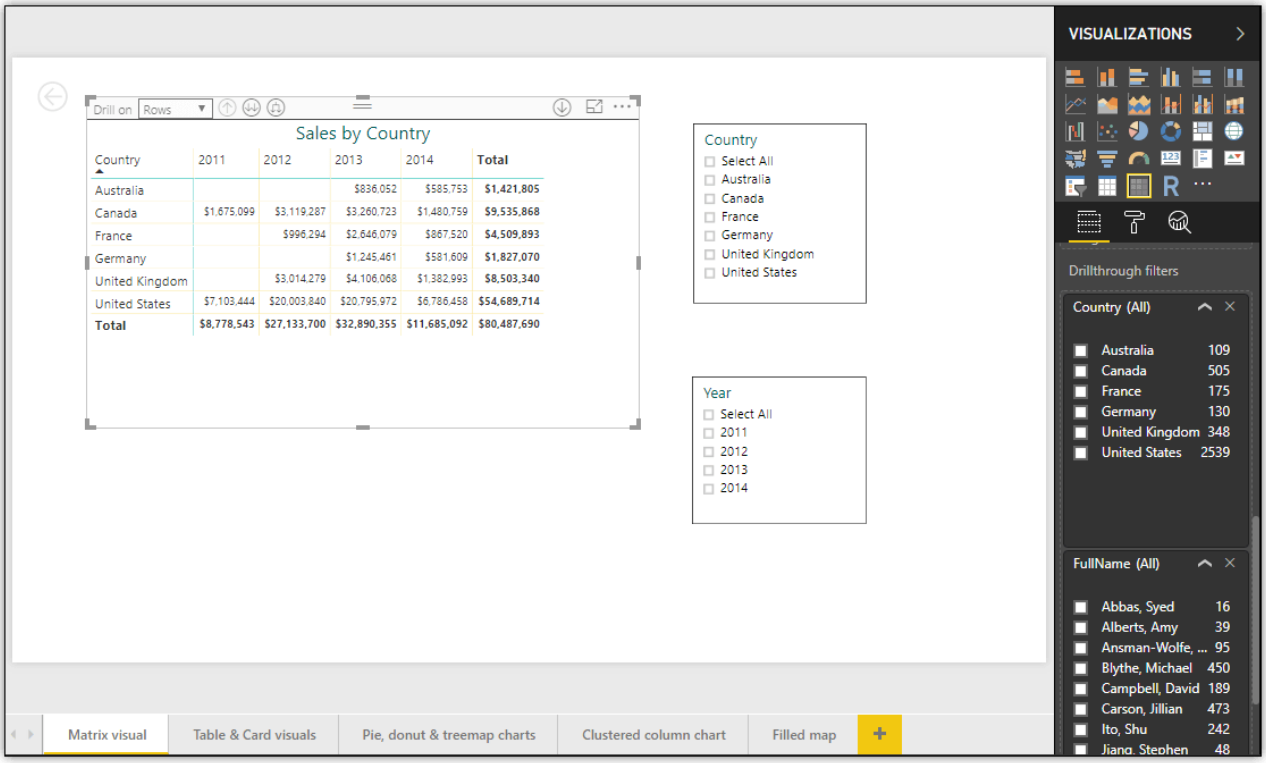


Figure 6. Updating the Matrix visual in Power BI Desktop

To add the drillthrough filters, select the *Matrix* visual and then drag the *Country* column from the *Fields* pane to the *Drillthrough* section of the *Visualizations* pane. Next, drag the *FullName* column to the *Drillthrough* section, just below the *Country* column.

Drillthrough filters make it possible for users to access one report based on values in another report. Because the *Country* and *FullName* columns have been added as drillthrough filters for the *Matrix* visual, users will be able to access the visual directly from other visuals that include *Country* or *FullName* values. Power BI automatically filters the *Matrix* visual by the selected country or sales rep name. (You'll see this in action in just a bit.)

After you've updated and saved the report in Power BI Desktop, you can publish it to the Power BI service just like you did before, except that you'll be prompted to verify that you're replacing the existing dataset, as shown in Figure 7. To confirm the update, click *Replace*.

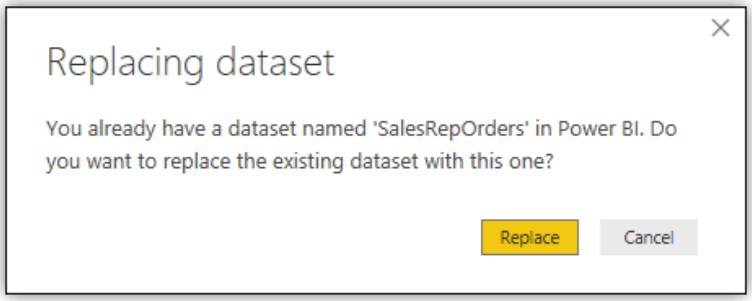


Figure 7. Republishing the *SalesRepOrders* report to the Power BI service

Although the wording of the *Replacing dataset* dialog box is specific to the *SalesRepOrders* dataset, the report itself is also updated in the Power BI service. You can verify this by returning to the Power BI service and viewing the *Matrix* report, which should now look similar to the one shown in Figure 8. (You might have to refresh the web page.) Notice that the *Drillthrough* section of the *Filters* tab now lists the *Country* and *FullName* columns as drillthrough filters.

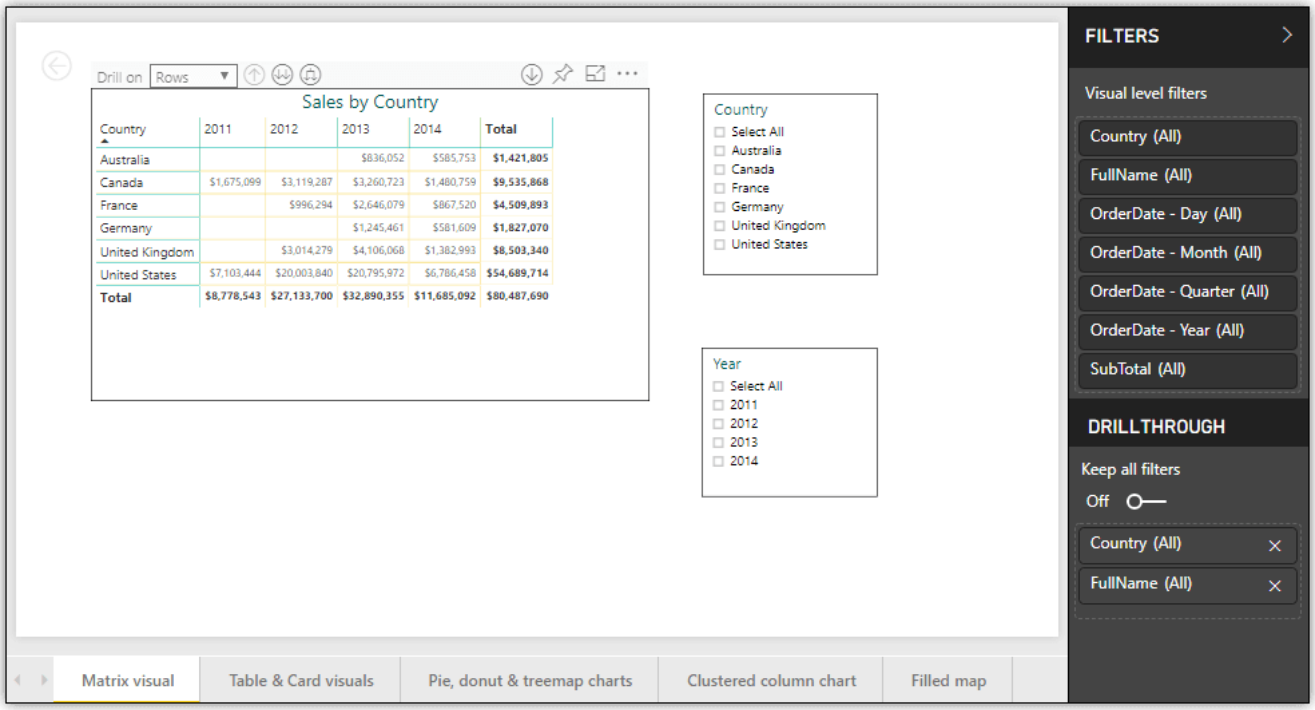


Figure 8. Viewing the updated report in Power BI

To test the drillthrough feature, go to the *Pie, donut and treemap chart* page, right-click the *Canada* section on the *Pie chart* visual, point to *Drillthrough,* and then click *Matrix visual,* as shown in Figure 9.

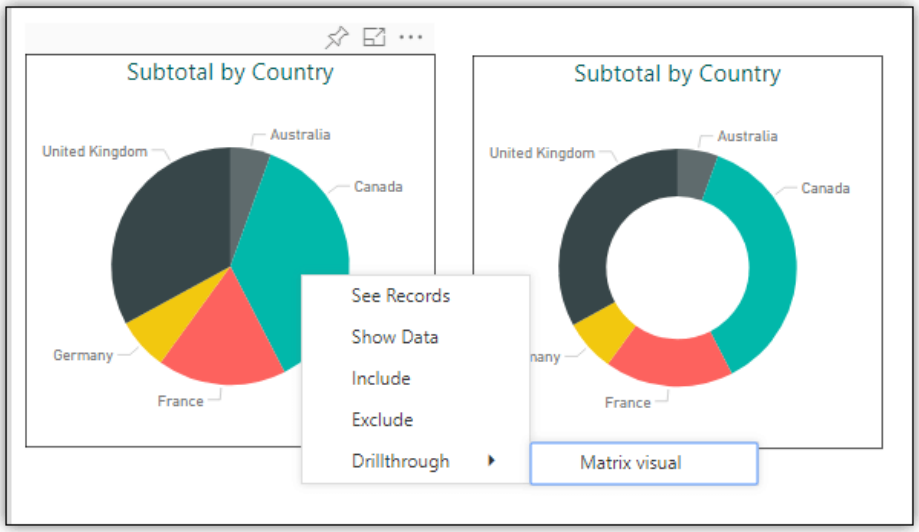
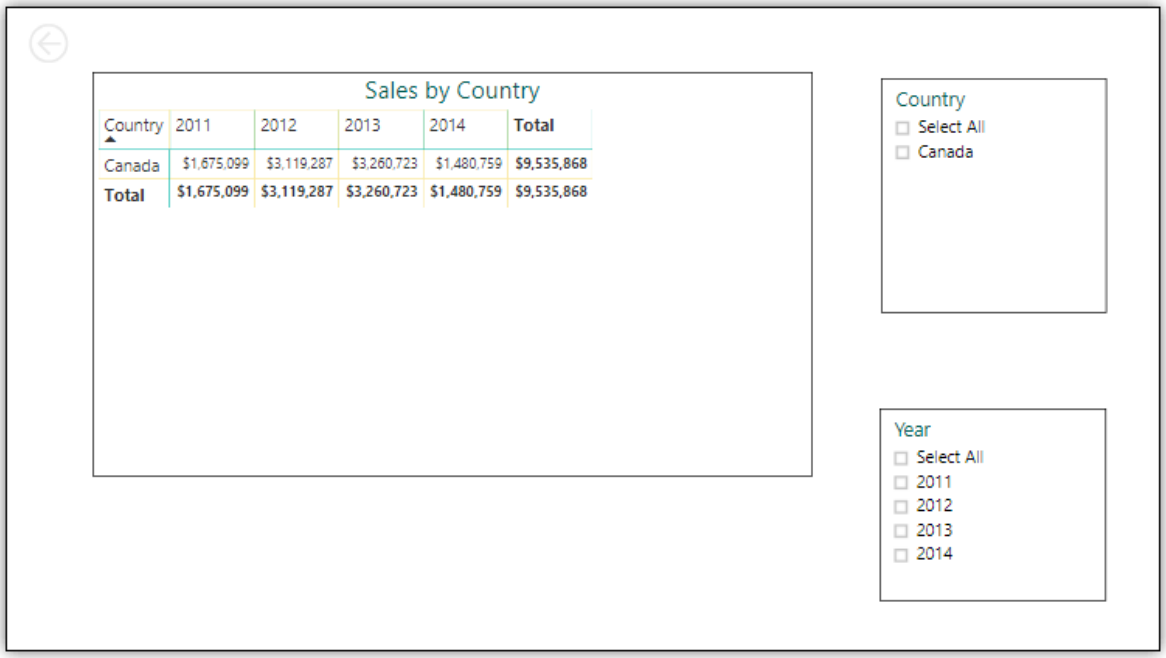


Figure 9. Drilling through to the *Matrix* visual in Power BI

When you click the *Matrix visual* option, Power BI takes you to the *Matrix visual* page, with the data filtered by *Canada,* as shown in Figure 10. Notice that *Canada* is also the only country listed in the *Country* slicer.

NOTE: If your *Matrix Visual* report shows quarters and/or months, select *Columns* from the *Drill on* list and click the up arrow. You may also need to remove the *Year* filter.



Country	2011	2012	2013	2014	Total
Canada	$1,675,099	$3,119,287	$3,260,723	$1,480,759	$9,535,868
Total	**$1,675,099**	**$3,119,287**	**$3,260,723**	**$1,480,759**	**$9,535,868**

Figure 10. Viewing Canadian sales in the *Matrix* visual

Next, you'll test drilling through to the *Matrix* visual from a sales rep value. First, you must clear the drillthrough filter on the *Matrix* visual. To do so, make sure the *Matrix* visual is still selected, expand the *Country* column in the *Drillthrough* section of the *Filters* tab, and then clear the *Canada* checkbox. Next, go to the *Clustered column chart* page. On the visual, right-click the *2012* bar for *Jillian Carson,* point to *Drillthrough,* and then click *Matrix visual*.

When the *Matrix* visual appears, the matrix will include only a row for the United States, the country where *Jillian Carson* resides. Click the double down-arrow button with the pop-up label that reads *Go to the next level in the hierarchy*. Jillian Carson's sales data should now be displayed in the *Matrix* visual, as shown in Figure 11.

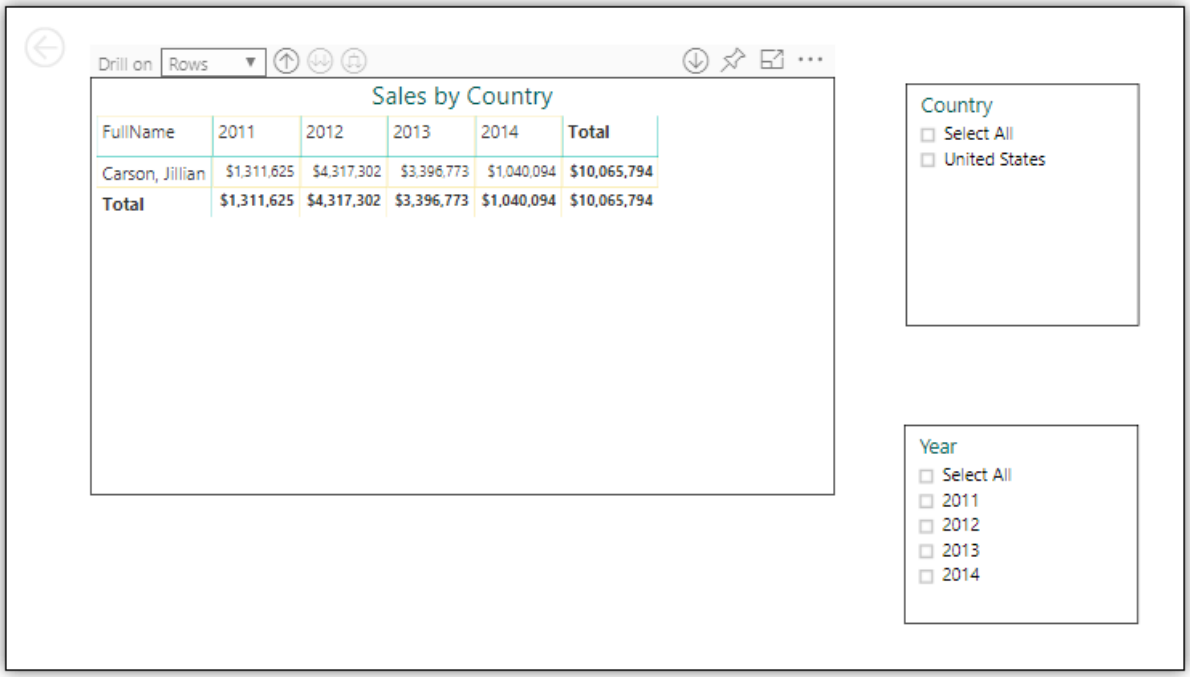


Figure 11. Viewing sales rep data in the *Matrix* visual

When you accessed the *Matrix* visual from the *Clustered column chart* visual, Power BI retained all sales data for Jillian Carson, even though you clicked the *2012* bar. However, you can limit the data to the year as well when accessing the *Matrix* visual through a drillthrough filter.

To demonstrate how this works, ensure that the *Matrix* visual is selected and then, in the *Drillthrough* section of the *Filters* tab, enable the *Keep all filters* option. Next, go to the *Clustered column chart* visual and again use the drillthrough feature to access the *Matrix* visual. This time, only the data for year *2012* is displayed, as shown in Figure 12.

Figure 12. Retaining filters when drilling through to a visual

As you can see, the *Matrix* visual and *Year* slicer now include only the year 2012. Not only does this demonstrate how the drillthrough capabilities work in Power BI, but also how you can update a visual in Power BI Desktop and republish the report file to the Power BI service, without having to take any steps within the Power BI service.

Pinning Reports to a Dashboard

When you publish a report file to the Power BI service, only the report and its datasets are added to the service. If you want to include report components on a dashboard, you must specifically pin them to the dashboard.

You can pin report items on a page-by-page basis directly within the report. For example, to add the *Clustered column chart* page to a dashboard, go to that page and click the *Pin Live Page* button at the top design surface. When the *Pin to dashboard* dialog box appears, you can choose to add the page to an existing dashboard or to create a new dashboard, as shown in Figure 13.

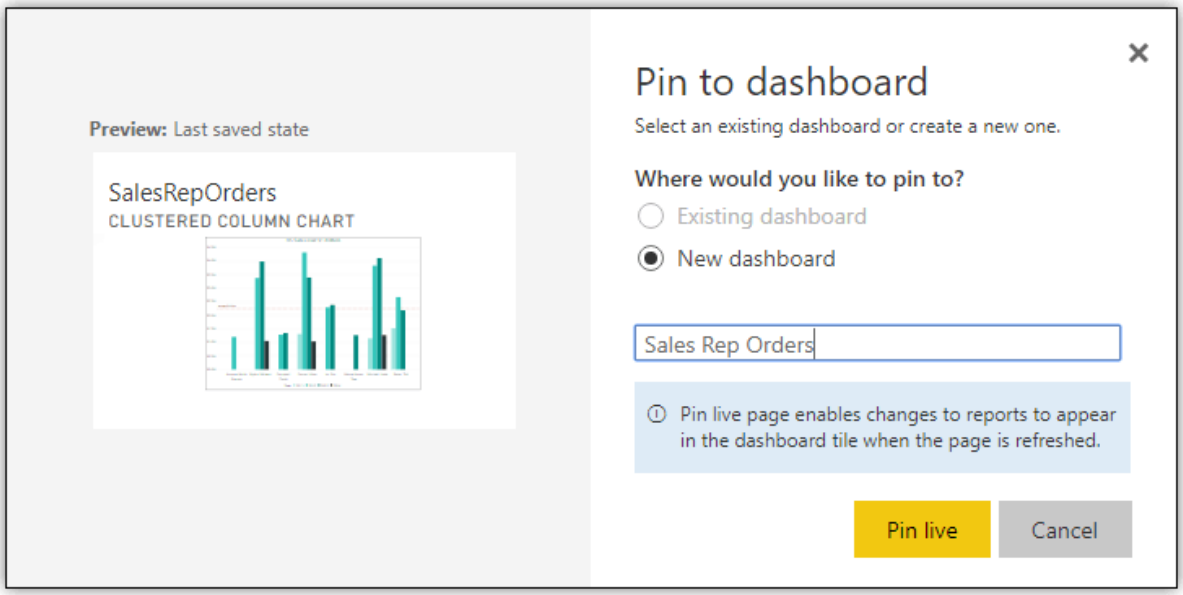


Figure 13. Pinning a report page to a new dashboard

If you choose the *Existing dashboard* option, you must then select the dashboard from a drop-down list. If choose the *New dashboard* option, you must provide a name for that dashboard. In Figure 13, a dashboard named *Sales Rep Orders* will be created.

After you select the dashboard or type the new name, click *Pin live*. When the *Pinned to dashboard* dialog box appears, click *Go to dashboard*, as shown in Figure 14. Later you'll see how you can set up your dashboard for a smartphone as well as a web page.

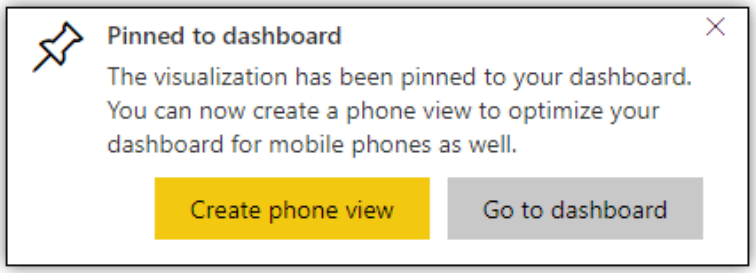


Figure 14. Creating a phone app version for the dashboard

By default, Power BI displays the dashboard in *Web view,* as shown in Figure 15. The report page is added as a tile to the dashboard. You can resize or reposition the tile, edit details about the tile, and carry out other steps.

Figure 15. Viewing the *Bar Chart* visual in web view

You can also make changes to the dashboard itself. For example, you can apply a different theme, refresh the tiles, or remove the *Ask a question* section. You can also switch to *Phone view*.

Phone view provides an approximation of what your dashboard will look like on a smartphone. To go to *Phone view,* click the *Web view* down arrow and then select *Phone view*. When you first select *Phone view,* you'll receive a message about viewing the tiles as they appear on a phone. Simply click *Continue*. You'll then be taken to *Phone view,* as shown in Figure 16.

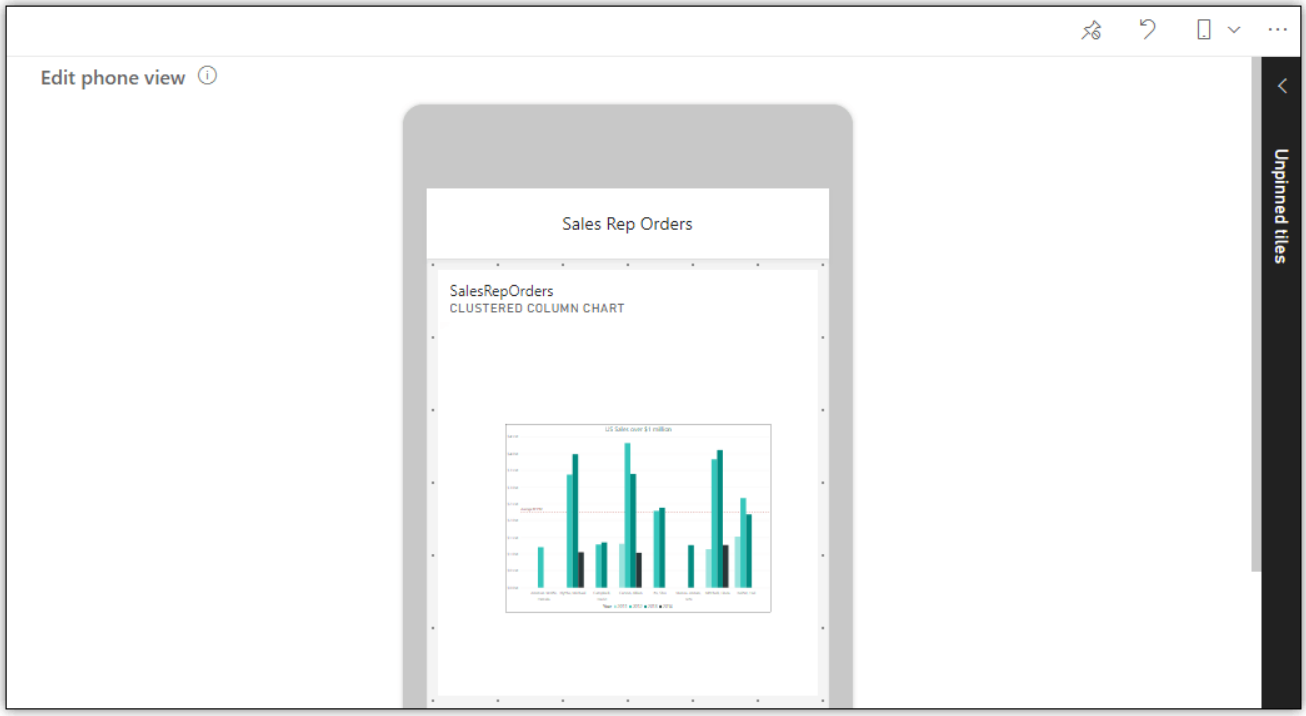


Figure 16. Viewing the *Bar Chart* visual in phone view

In *Phone view,* you can resize and reset tiles, but you can do little else. Even so, it should give you a good sense of how the dashboard will look on a smartphone. You can also view the dashboard in the Power BI app on a mobile device. For example, Figure 17 shows the *Clustered column chart* tile of the *Sales Rep Orders* dashboard on an iPhone.

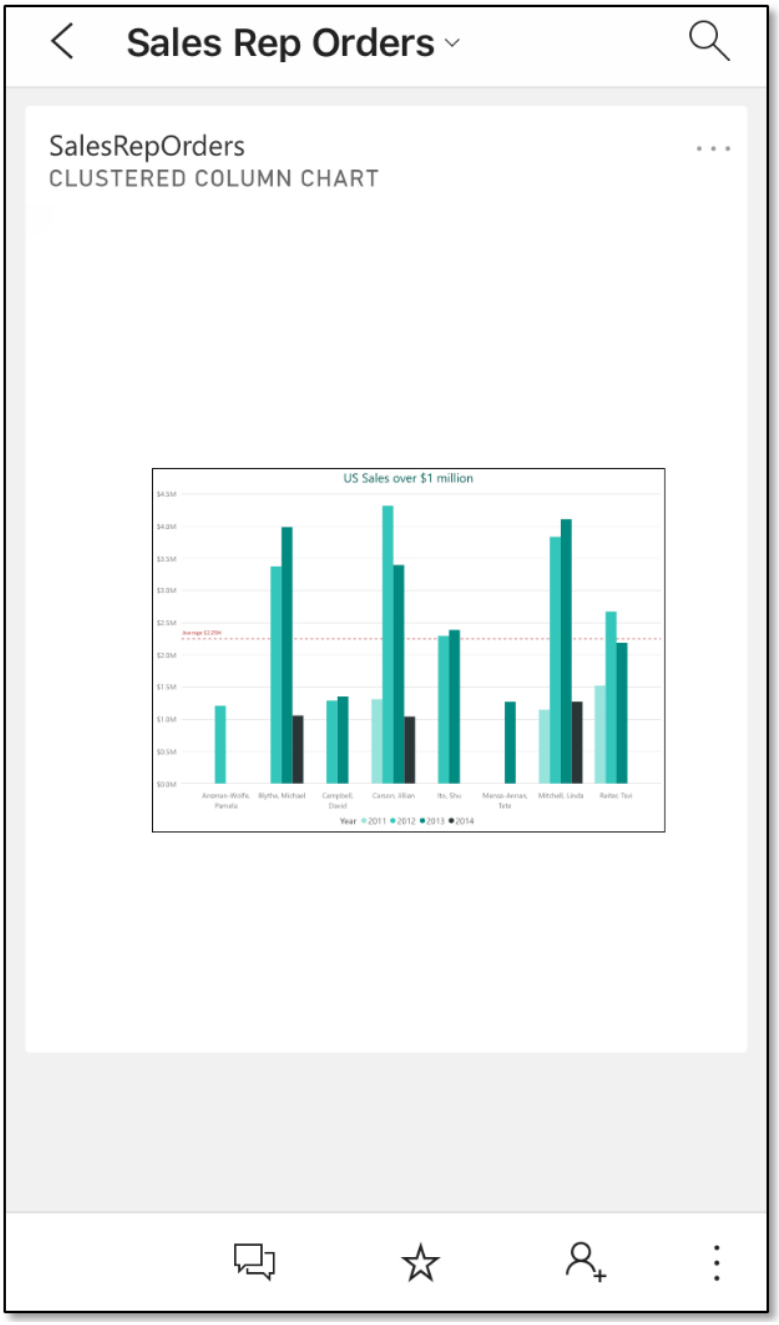


Figure 17. Viewing the dashboard report on an iPhone

There is, of course, a lot more you can do with the report and dashboard features in the Power BI service than what's been covered here. I encourage you to play around with the individual features as much as possible. The more time you spend with them, the better you'll be able to leverage their capabilities to effectively visualize data.

Creating Reports from a Dataset

In Power BI, you can also work directly with the datasets that you publish to the Power BI service. To access the *SalesRepOrders* dataset, select it in the *Datasets* category of the *My Workspace* section in the left pane. This will take you to

a workspace similar to *Report* view in Power BI Desktop, where you can create a report and add pages and visualizations. For example, Figure 18 shows a *Bar chart* visual that I created based on the *SalesRepOrders* dataset

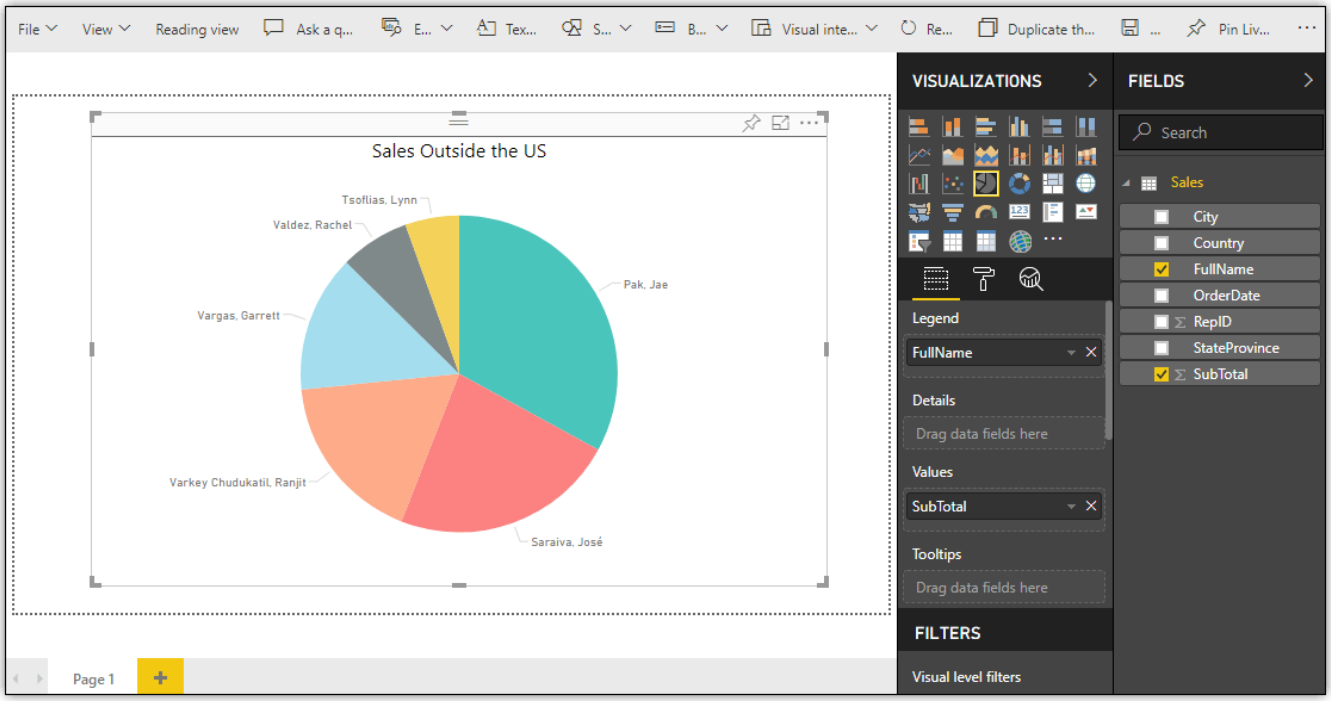


Figure 18. Creating a *Bar chart* visual based on the *SalesRepOrders* dataset

After you add the necessary pages and visualizations, you can save the report to the Power BI service, using the options available in the *File* menu. You can then access the report from the *Reports* category in the *My Workspace* section, just like the *SalesRepOrders* report.

For this example, I saved the report shown in Figure 18 as *SalesOutsideUS*. I then accessed the report from the *Reports* category, as shown in Figure 19. When you create a report in this way, you can pin the report pages to a dashboard, along with pages from other reports.

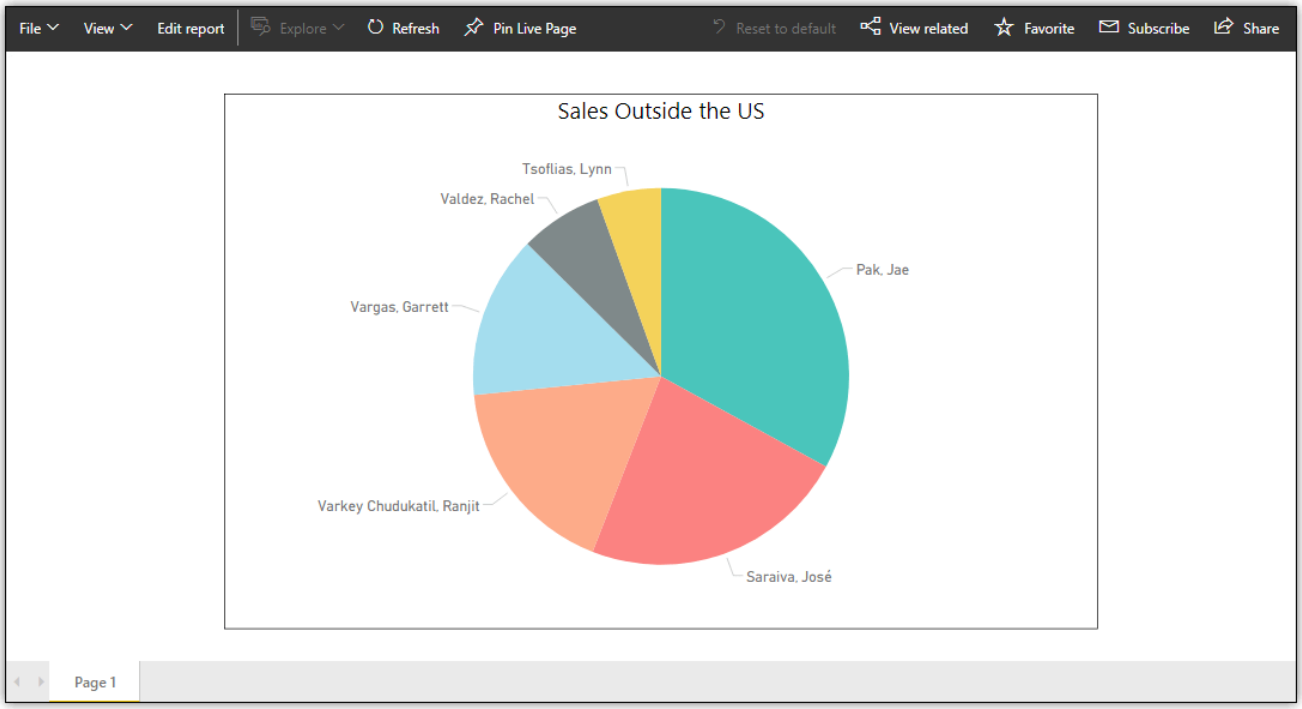


Figure 19. Viewing the *SalesOutsideUS* report in the Power BI interface

Being able to create reports directly from a published dataset provides you with even greater flexibility when working with the Power BI service. This can be especially beneficial when collaborating with other users to build reports and visualizations for your organization and other stakeholders.

Moving Beyond the Power BI Basic Service

The features covered in this article are all available as part of the Power BI Free service level. To use such features as sharing, collaboration, auditing, and auto-refresh, you (or your organization) must sign up for the Power BI Pro service. If your organization requires dedicated resources for deploying Power BI at scale, you'll need a Power BI Premium subscription.

Even at the Free service level, you can learn a lot about Power BI and its capabilities, especially when used in conjunction with Power BI Desktop. But Power BI is not just about the service and desktop application. The Power BI suite also includes the Power BI mobile apps, Power BI Report Server, and the Power BI API, with the promise of more components to come.

Microsoft also positions Power BI as part of something much larger called the Microsoft Power Platform (or Business Application Platform or whatever happens to be Microsoft's *branding de jour).* Not only does the platform include Power BI, but also PowerApps, Microsoft Flow, and perhaps Microsoft Stream, although it's not clear exactly which components are included or how they fit together. Even so, Power BI is a force to be reckoned with in its own right, and I suspect we'll be hearing a lot more about Power BI in the days to come.

Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

SQL Server Audit is a powerful feature that can help you comply with regulations such as HIPAA and SOX, but it's not easy to view the data collected. In this article, Robert Sheldon demonstrates how Power BI can be used to view and filter SQL Server Audit results.

The series so far:

1. Power BI Introduction: Tour of Power BI — Part 1
2. Power BI Introduction: Working with Power BI Desktop — Part 2
3. Power BI Introduction: Working with R Scripts in Power BI Desktop — Part 3
4. Power BI Introduction: Working with Parameters in Power BI Desktop — Part 4
5. Power BI Introduction: Working with SQL Server data in Power BI Desktop — Part 5
6. Power BI Introduction: Power Query M Formula Language in Power BI Desktop — Part 6
7. Power BI Introduction: Building Reports in Power BI Desktop — Part 7
8. Power BI Introduction: Publishing Reports to the Power BI Service — Part 8
9. Power BI Introduction: Visualizing SQL Server Audit Data — Part 9

Database teams trying to comply with regulations such as the Health Insurance Portability and Accountability Act of 1996 (HIPAA) or the Sarbanes-Oxley Act of 2002 (SOX) commonly use auditing as part of their compliance strategies to help track potential threats to stored data. For example, teams running SQL Server might turn to SQL Server Audit to log actions at both the server and database levels. SQL Server Audit is built into the database engine and, starting with SQL Server 2016, is available in all SQL Server editions.

If you're a DBA tasked with implementing SQL Server Audit, you'll find that setting up the auditing components is a relatively straightforward process. The more significant challenges often lie in trying to determine what user actions to audit, how to handle the potentially large amounts of audit data, and what tools to use to monitor and review that data.

Although these are all important considerations, this article is concerned primarily with the last one—monitoring and reviewing the audit data. SQL Server makes it easy to collect the data but provides no mechanisms for working with that data in a meaningful way, other than to review it manually.

One way to address this issue is to bring in a third-party solution such as EventTracker to consolidate, manage, and monitor the audit data, but this comes with additional licensing and implementation considerations.

Another approach is to use Power BI to create reports that provide quick visual insights into the data. Although Power BI is not designed for auditing and alerting to the same degree as more robust log management tools, it provides a relatively simple

method for tracking user behavior. Best of all, the basic Power BI service is free, as is Power BI Desktop, a downloadable tool for transforming and visualizing different types of data.

In this article, I demonstrate how you can use Power BI to work with SQL Server Audit data, walking you through the steps necessary to set up a test environment, generate sample audit data, pull that data into Power BI Desktop, and create a report that contains tables and visualizations. Keep in mind, however, that SQL Server Audit and Power BI Desktop are both powerful solutions that support far more capabilities than can be covered in a single article. The information here should at least provide you with a conceptual overview of how the tools can be used together and what it takes to get started using Power BI Desktop to review audit data.

Setting Up a Test Environment

To prepare your environment for the examples in this article, you should first create the `ImportSales` test database, which contains one schema, `Sales`, and one table, `Sales.Customers`. You can then populate the table with data from the `Sales.Customers` table in the `WideWorldImporters` database. For this article, all audited actions are limited to `Customers` table in the `ImportSales` database.

To create the `ImportSales` database, run the following T-SQL code:

```
USE master;
GO
DROP DATABASE IF EXISTS ImportSales;
GO
CREATE DATABASE ImportSales;
GO
USE ImportSales;
GO
CREATE SCHEMA Sales;
GO
CREATE TABLE Sales.Customers(
  CustID INT IDENTITY PRIMARY KEY,
  Customer NVARCHAR(100) NOT NULL,
  Contact NVARCHAR(50) NOT NULL,
  Email NVARCHAR(256) NULL,
  Phone NVARCHAR(20) NULL,
  Category NVARCHAR(50) NOT NULL);
GO
INSERT INTO Sales.Customers(Customer, Contact, Email, Phone, Category)
SELECT c.CustomerName, p.FullName, p.EmailAddress,
  p.PhoneNumber, cc.CustomerCategoryName
FROM WideWorldImporters.Sales.Customers c
  INNER JOIN WideWorldImporters.Application.People p
    ON c.PrimaryContactPersonID = p.PersonID
  INNER JOIN WideWorldImporters.Sales.CustomerCategories cc
    ON c.CustomerCategoryID = cc.CustomerCategoryID;
GO
```

If you don't have the `WideWorldImporters` database installed, you can populate the `Customers` table with your own data. If you want to use a different table and database for these examples, skip the T-SQL statements above and use a database and table that best suit your needs (and are not in a production environment, of course). Just be sure to replace any references to the `ImportSales` database or `Customers` table in the subsequent examples.

The next step is to create an audit object at the SQL Server instance level and a database audit specification at the `ImportSales` database level. The audit object serves as a container for organizing the server and database audit settings and for delivering the final audit logs. For this article, you'll be saving the audit data to a local folder, but know that SQL Server Audit also lets you save the data to the Windows Application log or Security log.

A database audit specification is created at the database level and is associated with an audit object, which means that the audit object needs to exist before you can create the database audit specification. The specification determines what actions should be audited at the database level. You can also create a server audit specification for auditing actions at the server level, but for this article, you need only the database audit specification.

To create both the audit object and database audit specification, run the following T-SQL code:

```
USE master;
GO
CREATE SERVER AUDIT ImportSalesAudit
TO FILE (FILEPATH = 'C:\DataFiles\audit\');
GO
ALTER SERVER AUDIT ImportSalesAudit
WITH (STATE = ON);
GO
USE ImportSales;
GO
```

```
CREATE DATABASE AUDIT SPECIFICATION ImportSalesDbSpec
FOR SERVER AUDIT ImportSalesAudit
ADD (SCHEMA_OBJECT_CHANGE_GROUP),
ADD (SELECT, INSERT, UPDATE, DELETE
  ON Object::Sales.Customers BY public)
WITH (STATE = ON);
GO
```

The `CREATE SERVER AUDIT` statement creates an audit object named `ImportSalesAudit` that saves the audit data to the *C:\DataFiles\Audit* folder. You must then run an `ALTER SERVER AUDIT` statement as a separate step to set the `STATE` property to `ON`.

Next comes the `CREATE DATABASE AUDIT SPECIFICATION` statement, which defines a specification named `ImportSalesDbSpec`. The specification includes two `ADD` clauses. The first `ADD` clause specifies the action group `SCHEMA_OBJECT_CHANGE_GROUP`, which audits all `CREATE`, `ALTER`, and `DROP` statements issued against any schema objects in the database. Because this is a group action, it must be specified separately from individual actions, such as those in the second `ADD` clause.

The second `ADD` clause specifies four individual actions:

- The `SELECT` action audits all `SELECT` statements.
- The `INSERT` action audits all `INSERT` statements.
- The `UPDATE` action audits all `UPDATE` statements.
- The `DELETE` action audits all `DELETE` statements.

The `ON` subclause in the second `ADD` clause points to the `Customers` table, which means that the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` actions are all specific to that table. In addition, because the `BY` subclause specifies the `public` login, the auditing applies to all users.

Under normal circumstances, you would likely be auditing many more users and actions than you've done here, but this is enough to demonstrate the basic principles behind using Power BI to review audit data.

With the auditing structure in place, run the following T-SQL code to create three test user accounts within the `ImportSales` database, assigning a different set of permissions to each user:

```
CREATE USER User01 WITHOUT LOGIN;
GRANT ALTER, SELECT, INSERT, DELETE, UPDATE
ON OBJECT::Sales.Customers TO user01;
GO
CREATE USER User02 WITHOUT LOGIN;
GRANT SELECT, INSERT, DELETE, UPDATE
ON OBJECT::Sales.Customers TO user02;
GO
CREATE USER User03 WITHOUT LOGIN;
GRANT SELECT
ON OBJECT::Sales.Customers TO user03;
GO
```

The user accounts are created without logins to keep the testing simple. For the same reason, all the granted permissions are specific to the `Customers` table, with access defined as follows:

- `User01` can access and modify all data within the table as well as update the table's definition.
- `User02` can access and modify all data within the table but is not permitted to update the table's definition.
- `User03` can access all data within the table but is not permitted to modify any of the data or update the table's definition.

You can set up the test users and their permissions any way you like, so that you have them in place when you're ready to generate audit data to use in Power BI Desktop.

Generating Test Audit Data

To generate the audit data, you should run a series of data manipulation language (DML) statements and data definition language (DDL) statements against the `Customers` table, running them within the execution context of the three database user accounts. The easiest way to do this is to use an `EXECUTE AS` statement to specify the user context, run one or more statements, and then run a `REVERT` statement to switch back to the original user.

You can run whatever DML and DDL statements you want, as long as you test each account and the permissions assigned to that account. On my system, I ran several sets of T-SQL statements, most of them multiple times to generate a reasonable amount of data. If you want to use the same statements that I used, start with the following DML statements, running them under the `User01` account:

```
EXECUTE AS USER = 'User01';
SELECT * FROM Sales.Customers;
INSERT INTO Sales.Customers
  (Customer, Contact, Email, Phone, Category)
  VALUES('Wingtip Toys (Eugene, OR)', 'Flora Olofsson',
    'flora@wingtiptoys.com', '(787) 555-0100', 'Gift Store');
DECLARE @LastID INT = (SELECT SCOPE_IDENTITY())
UPDATE Sales.Customers SET Category = 'Novelty Shop'
WHERE CustID = @LastID;
DELETE Sales.Customers WHERE CustID = @LastID;
REVERT;
GO
```

Be sure to run these statements multiple times, according to how much audit data you want available. I ran the DML statements about five times and the DDL statements either one or two times.

After you run the preceding DML statements, run the following statements to add a column to the `Customers` table, again as `User01`:

```
EXECUTE AS USER = 'User01';
ALTER TABLE Sales.Customers
ADD Status BIT NOT NULL DEFAULT(1);
REVERT;
GO
```

Next, repeat the same DML statements for `User02`, again running the block of statements multiple times:

```
EXECUTE AS USER = 'User02';
SELECT * FROM Sales.Customers;
INSERT INTO Sales.Customers
  (Customer, Contact, Email, Phone, Category)
  VALUES('Tailspin Toys (Bainbridge Island, WA)', 'Kanti Kotadia',
    'kanti@tailspintoys.com', '(303) 555-0100', 'Gift Store');
DECLARE @LastID INT = (SELECT SCOPE_IDENTITY())
UPDATE Sales.Customers SET Category = 'Novelty Shop'
WHERE CustID = @LastID;
DELETE Sales.Customers WHERE CustID = @LastID;
REVERT;
GO
```

Now try to add another column to the `Customers` table, but this time as `User02`, using the following T-SQL:

```
EXECUTE AS USER = 'User02';
ALTER TABLE Sales.Customers
ADD LastUpdated DATETIME NOT NULL DEFAULT(GETDATE());
REVERT;
GO
```

The statement should generate an error because the `User02` account does not have the proper permissions to modify the table definition. Be aware, however, that if you run one or more T-SQL statements under the context of a specific user and one of the statements fails, the `REVERT` statement will not run, in which case, you must rerun the REVERT statement, without the other statements, to ensure that you close that execution context. (A better approach, of course, is to add the proper logic to your code to ensure that the `REVERT` statement always runs.)

Next, run the same DML statements as before, but under the `User03` account:

```
EXECUTE AS USER = 'User03';
SELECT * FROM Sales.Customers;
INSERT INTO Sales.Customers
  (Customer, Contact, Email, Phone, Category)
  VALUES('Tailspin Toys (Bainbridge Island, WA)', 'Kanti Kotadia',
    'kanti@tailspintoys.com', '(303) 555-0100', 'Gift Store');
DECLARE @LastID INT = (SELECT SCOPE_IDENTITY())
UPDATE Sales.Customers SET Category = 'Novelty Shop'
WHERE CustID = @LastID;
DELETE Sales.Customers WHERE CustID = @LastID;
REVERT;
GO
```

In this case, the `INSERT`, `UPDATE`, and `DELETE` statements each return an error because of the lack of permissions, which means you'll once again need to run the `REVERT` statement separately. The same goes for the following `ALTER TABLE` statement, which also returns an error:

```
EXECUTE AS USER = 'User03';
ALTER TABLE Sales.Customers
ADD LastUpdated DATETIME NOT NULL DEFAULT(GETDATE());
REVERT;
```

Again, you can run whatever DML and DDL statements you like to generate the test audit statements. The idea is to have enough data to use in Power BI Desktop.

Connecting to SQL Server Audit Data in Power BI Desktop

When connecting to SQL Server within Power BI Desktop, you can retrieve the data from specific tables and views, or you can run a query that returns exactly the data you need from multiple tables and views. With a query, you can also use system functions such as `sys.fn_get_audit_file`, a table-valued function that returns data from SQL Server Audit log files. (You cannot use this function to retrieve data from the Application or Security log.)

For this article, you will use the function in the following `SELECT` statement to return the user account, action, success status, T-SQL statement, and event time of each logged event:

```
SELECT f.database_principal_name [User Acct],
  (CASE
    WHEN a.name = 'STATEMENT ROLLBACK' THEN 'ROLLBACK'
    ELSE a.name
  END) [User Action],
  (CASE
    WHEN f.succeeded = 1 THEN 'Succeeded'
    ELSE 'Failed'
  END) [Succeeded],
  f.statement [SQL Statement],
  f.event_time [Date/Time]
FROM sys.fn_get_audit_file
  ('C:\DataFiles\audit\ImportSalesAudit_*.sqlaudit',
   default, default) f
  INNER JOIN (SELECT DISTINCT action_id, name
  FROM sys.dm_audit_actions) a
    ON f.action_id = a.action_id
WHERE f.database_principal_name IN ('User01', 'User02', 'User03')
```

The statement joins the `sys.fn_get_audit_file` function to the `sys.dm_audit_actions` function to return an action's full name, rather than its abbreviation. The statement also limits the results to the three test user accounts.

You'll use this `SELECT` statement when setting up your SQL Server connection in Power BI Desktop. To configure the connection, create a new report in Power BI Desktop, click the *Get Data* down arrow on the *Home* ribbon, and then click *SQL Server*. When the *SQL Server database* dialog box appears, click the *Advanced options* arrow to expand the dialog box, as shown in Figure 1.

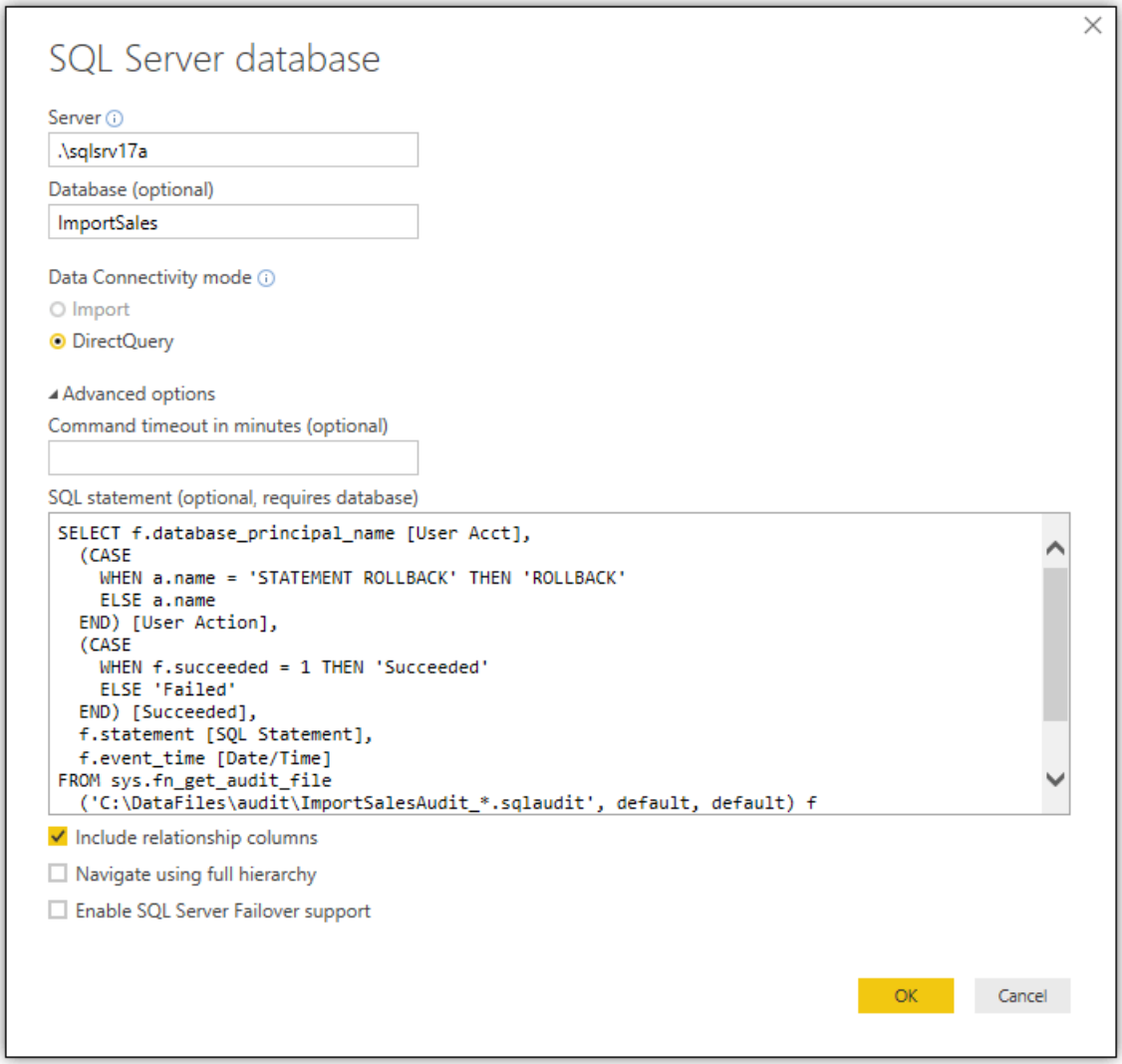


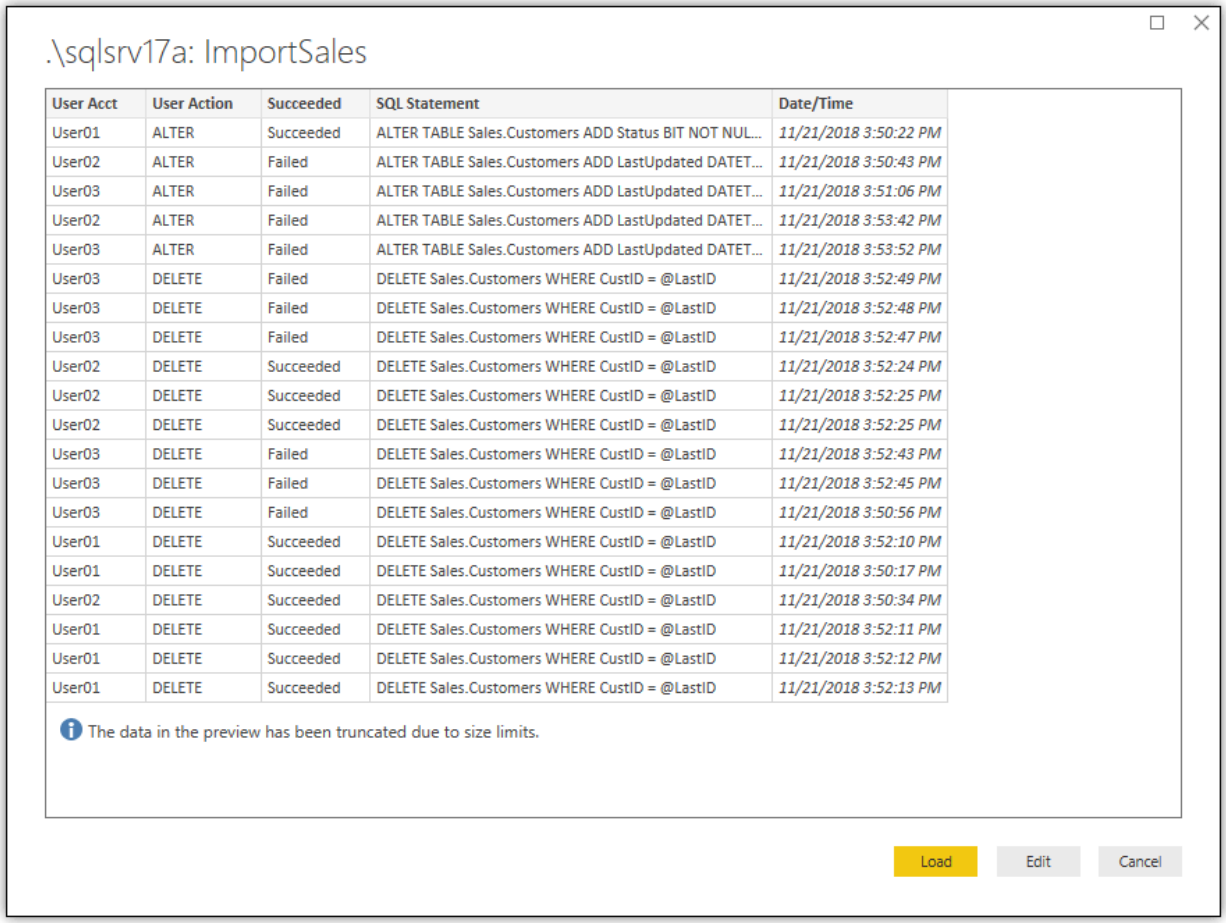
Figure 1. Defining a SQL Server connection in Power BI Desktop

To configure the connection:

1. Type the SQL Server instance in the *Server* text box.
2. Type the name of the database, `ImportSales`, in the *Database* text box.
3. Select an option in the *Data Connectivity* mode section. I chose *DirectQuery*.
4. Type or paste the `SELECT` statement into the *SQL statement* text box.

When you select *DirectQuery,* no data is imported or copied into Power BI Desktop. Instead, Power BI Desktop queries the underlying data source whenever you create or interact with a visualization, ensuring that you're always viewing the most current data. Be aware, however, that if you plan to publish your report to the Power BI service, you'll need to set up a gateway connection that enables the service to retrieve the source data. Also, note that you cannot create a Power BI Desktop report that includes both a *DirectQuery* SQL Server connection and an *Import* SQL Server connection. You must choose one or the other.

After you've configured the connection in the *SQL Server database* dialog box, click *OK*. This launches a preview window, where you can view a sample of the audit data, as shown in Figure 2.



.\sqlsrv17a: ImportSales

User Acct	User Action	Succeeded	SQL Statement	Date/Time
User01	ALTER	Succeeded	ALTER TABLE Sales.Customers ADD Status BIT NOT NUL...	11/21/2018 3:50:22 PM
User02	ALTER	Failed	ALTER TABLE Sales.Customers ADD LastUpdated DATET...	11/21/2018 3:50:43 PM
User03	ALTER	Failed	ALTER TABLE Sales.Customers ADD LastUpdated DATET...	11/21/2018 3:51:06 PM
User02	ALTER	Failed	ALTER TABLE Sales.Customers ADD LastUpdated DATET...	11/21/2018 3:53:42 PM
User03	ALTER	Failed	ALTER TABLE Sales.Customers ADD LastUpdated DATET...	11/21/2018 3:53:52 PM
User03	DELETE	Failed	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:49 PM
User03	DELETE	Failed	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:48 PM
User03	DELETE	Failed	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:47 PM
User02	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:24 PM
User02	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:25 PM
User02	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:25 PM
User03	DELETE	Failed	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:43 PM
User03	DELETE	Failed	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:45 PM
User03	DELETE	Failed	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:50:56 PM
User01	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:10 PM
User01	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:50:17 PM
User02	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:50:34 PM
User01	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:11 PM
User01	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:12 PM
User01	DELETE	Succeeded	DELETE Sales.Customers WHERE CustID = @LastID	11/21/2018 3:52:13 PM

The data in the preview has been truncated due to size limits.

Load Edit Cancel

Figure 2. Verifying the SQL Server Audit data

If everything looks as you would expect, click *Load* to make the data available to Power BI Desktop, where you can use the data to add tables or visualizations. If you selected *DirectQuery,* you're loading only the table schema into Power BI Desktop, with no source data imported until it is needed.

As noted earlier, you're not limited to saving the audit data to log files. You can also save the data to the Application or Security log, but that means taking extra steps to get the data out of the logs and into a digestible format, such as a .csv file.

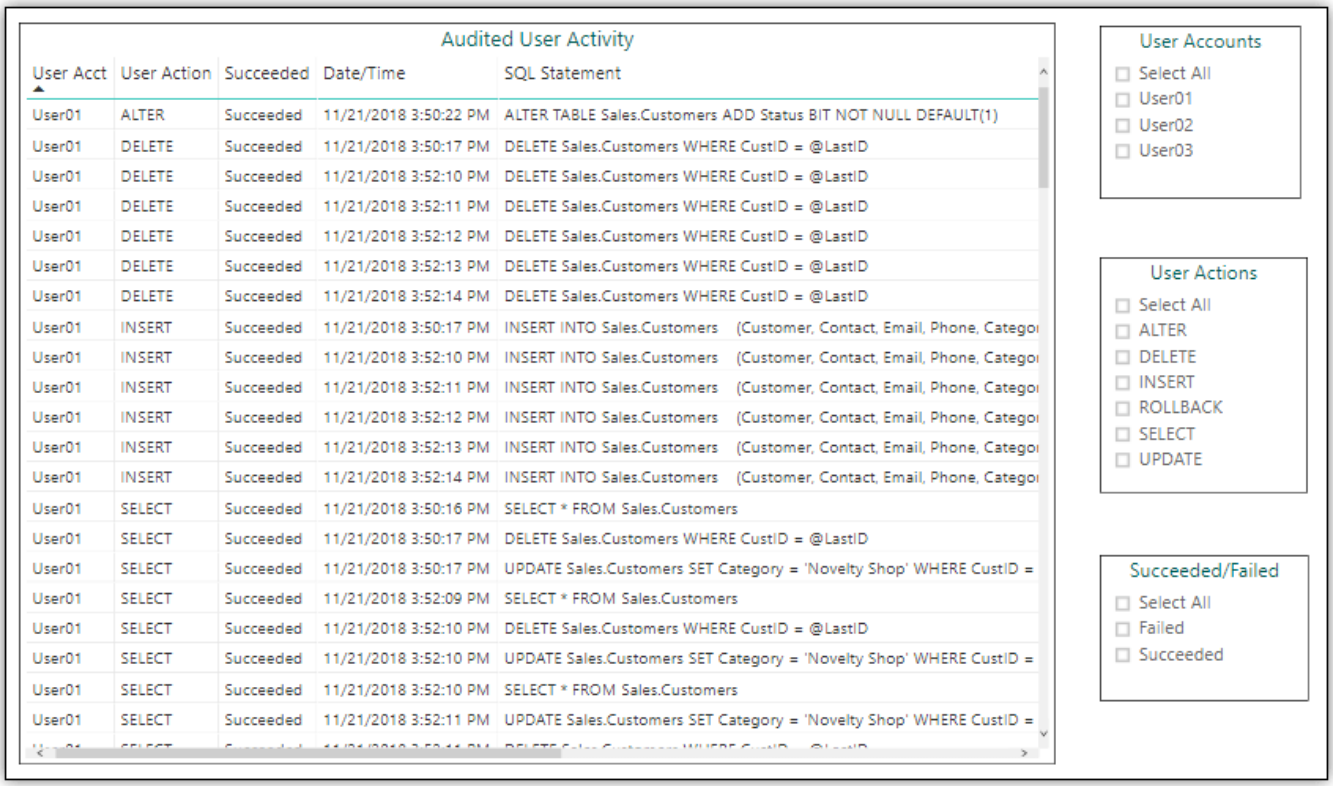
For example, you can export the data from Windows Event Viewer or the Log File Viewer in SQL Server Management Studio (SSMS), but the resulting format can be difficult to work with, and you could end up with much more data than you need. Another approach is to use PowerShell to retrieve the log data, creating a script that can be scheduled to run automatically. In this way, you have far more control over the output, although there's still some work involved to get it right.

Regardless of how you make the data available to Power BI Desktop, the more important consideration is data security. The Security log might be the safest approach initially, but once you export the data to files, you're up against the same issues you face when sending the data directly to log files. The data in those files needs to be fully protected at all times both at-rest and in-motion. There would be little point in implementing an audit strategy to comply with regulatory standards if the auditing process itself puts the data at risk.

Once you've made the test audit data available to Power BI Desktop, you can create reports that present the data in ways that offer different insights. Throughout the rest of the article, I show you several approaches that I've tried, using both tables and visualizations, in order to demonstrate ways you can present SQL Server Audit data. For details on how to actually create these components, refer to Part 7 of this series, Building Reports in Power BI Desktop.

Adding a Table to Your Report

One useful approach to presenting information is to make at least some of the data available in tables. For example, Figure 3 shows a table that includes all the audit data I pulled into Power BI Desktop. You can, of course, filter the data however you need to, such as with Slicers, depending on your specific audit strategy. The auditing that's implemented for this article is very simple in comparison to the types and amounts of data that you will likely be generating.



User Acct	User Action	Succeeded	Date/Time	SQL Statement
User01	ALTER	Succeeded	11/21/2018 3:50:22 PM	ALTER TABLE Sales.Customers ADD Status BIT NOT NULL DEFAULT(1)
User01	DELETE	Succeeded	11/21/2018 3:50:17 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	DELETE	Succeeded	11/21/2018 3:52:10 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	DELETE	Succeeded	11/21/2018 3:52:11 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	DELETE	Succeeded	11/21/2018 3:52:12 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	DELETE	Succeeded	11/21/2018 3:52:13 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	DELETE	Succeeded	11/21/2018 3:52:14 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	INSERT	Succeeded	11/21/2018 3:50:17 PM	INSERT INTO Sales.Customers (Customer, Contact, Email, Phone, Catego
User01	INSERT	Succeeded	11/21/2018 3:52:10 PM	INSERT INTO Sales.Customers (Customer, Contact, Email, Phone, Catego
User01	INSERT	Succeeded	11/21/2018 3:52:11 PM	INSERT INTO Sales.Customers (Customer, Contact, Email, Phone, Catego
User01	INSERT	Succeeded	11/21/2018 3:52:12 PM	INSERT INTO Sales.Customers (Customer, Contact, Email, Phone, Catego
User01	INSERT	Succeeded	11/21/2018 3:52:13 PM	INSERT INTO Sales.Customers (Customer, Contact, Email, Phone, Catego
User01	INSERT	Succeeded	11/21/2018 3:52:14 PM	INSERT INTO Sales.Customers (Customer, Contact, Email, Phone, Catego
User01	SELECT	Succeeded	11/21/2018 3:50:16 PM	SELECT * FROM Sales.Customers
User01	SELECT	Succeeded	11/21/2018 3:50:17 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	SELECT	Succeeded	11/21/2018 3:50:17 PM	UPDATE Sales.Customers SET Category = 'Novelty Shop' WHERE CustID =
User01	SELECT	Succeeded	11/21/2018 3:52:09 PM	SELECT * FROM Sales.Customers
User01	SELECT	Succeeded	11/21/2018 3:52:10 PM	DELETE Sales.Customers WHERE CustID = @LastID
User01	SELECT	Succeeded	11/21/2018 3:52:10 PM	UPDATE Sales.Customers SET Category = 'Novelty Shop' WHERE CustID =
User01	SELECT	Succeeded	11/21/2018 3:52:10 PM	SELECT * FROM Sales.Customers
User01	SELECT	Succeeded	11/21/2018 3:52:11 PM	UPDATE Sales.Customers SET Category = 'Novelty Shop' WHERE CustID =

Figure 3. Adding a table and slicers to a report page

In addition to the table, I added three slicers to the report page for filtering the data by users, actions, and successes and failures. For example, Figure 4 shows the data I generated on my system after I filtered the information by the `User03` account and the `Failed` status.

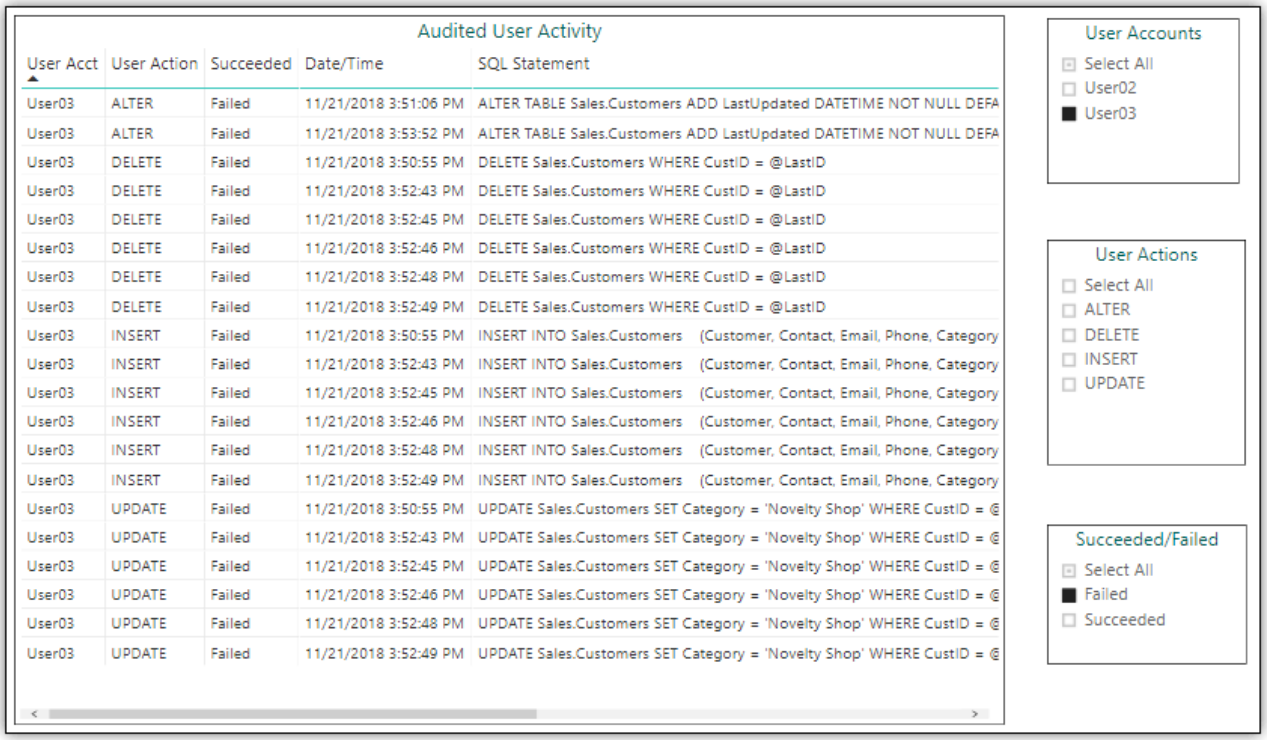


Figure 4. Using slicers to filter data in a table

When you apply a filter, Power BI updates the data in the table and slicers, based on the selected values. In this way, you have a quick and easy way to access various categories of data, without needing to manually weed through gobs of data or generate a new T-SQL statement each time you want to focus on different information.

Adding a Matrix to Your Report

Another effective way to provide quick insights into the data is to add a matrix that summarizes the data. For example, Figure 5 shows a matrix that displays the number of actions per user and action type. (A `ROLLBACK` statement is issued when a primary statement fails, such as when a user does not have the permissions necessary to run a statement.)

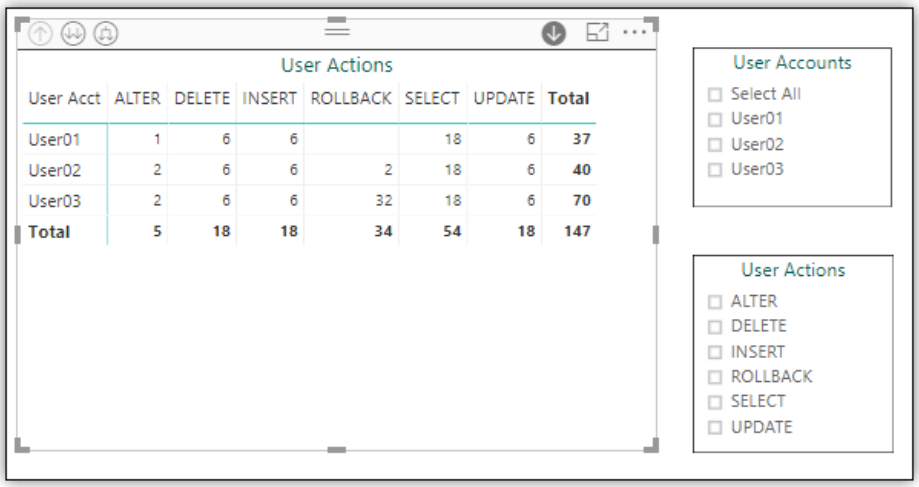
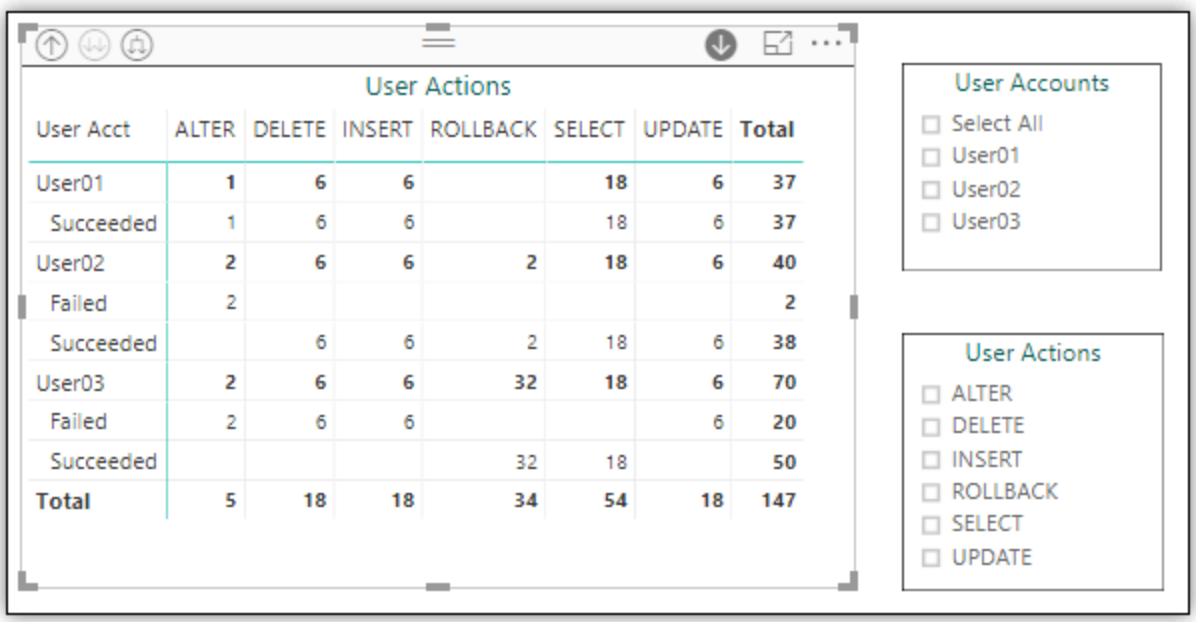


Figure 5. Adding a matrix and slicers to a report page

On the same report page as the matrix, I added two slicers, one for user accounts and the other for user actions, again allowing me to easily filter the data as needed.

What is especially useful about the matrix is that you can set it up to drill down into the data. In this case, the matrix allows you to drill to the number of actions that have succeeded or failed. For example, Figure 6 shows the matrix after drilling down into all users, although it's also possible to drill down into specific users.



User Actions

User Acct	ALTER	DELETE	INSERT	ROLLBACK	SELECT	UPDATE	Total
User01	1	6	6		18	6	37
Succeeded	1	6	6		18	6	37
User02	2	6	6	2	18	6	40
Failed	2						2
Succeeded		6	6	2	18	6	38
User03	2	6	6	32	18	6	70
Failed	2	6	6			6	20
Succeeded				32	18		50
Total	5	18	18	34	54	18	147

Figure 6. Drilling into a matrix

When setting up the drill-down categories in a matrix, you can choose the order in which to present each layer, depending on the type of data and its hierarchical nature. For example, this matrix can also be configured to drill down further into the T-SQL statements associated with each category.

Adding Visualizations to Your Report

Power BI Desktop supports a wide variety of visualizations, and you can import even more. You should use whichever visualizations help you and other users best understand the types of audit data you're collecting. For example, Figure 7 shows a report page with three types of visualizations, each providing a different perspective into the audit data (keeping in mind that you're working with a limited sample size).

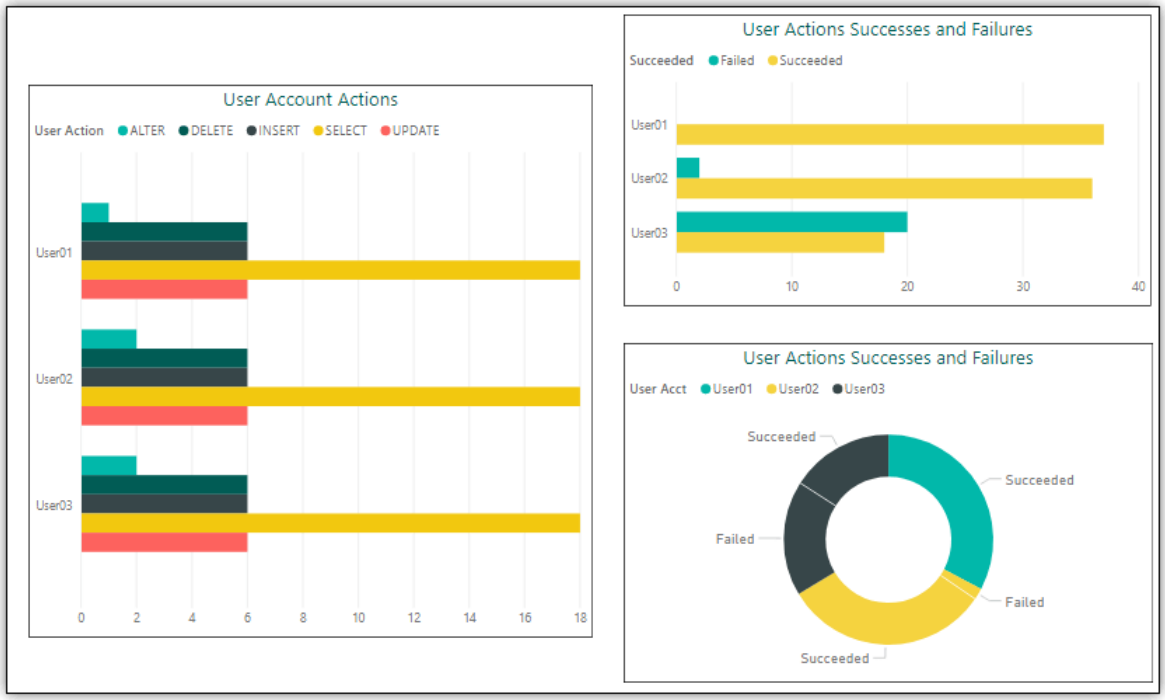


Figure 7. Adding visualizations to a report page

For all three visualizations, I filtered out the `ROLLBACK` actions, so the visualizations reflect only the statements that were initiated by the users, rather than those generated as a response to their actions. The visualizations on the left and top-right are clustered bar charts. The clustered bar chart lets you group related data in various ways to provide different perspectives.

For example, the clustered bar chart on the left groups the data by user, and for each user, provides a total for each action type. In this case, the DML actions show the same totals across all three user accounts because I ran the statements the same number of times. Even so, the visualization still demonstrates how you can use the clustered bar chart to provide a quick overview of the data from different perspectives, in this case, the types of statements each user has tried to run.

The clustered bar chart at the top right also groups the data by user, but then provides a total of statement successes and failures for each account, making it possible to see which users might be attempting to run statements that they're not permitted to run.

The bottom right visualization is a basic donut chart. The visualization presents the same data as the clustered bar chart above it, but in a different way. If you hover over one of the elements, for example, you'll get the percentage of total actions. What all this points to is that you can try out different visualizations against the same data to see which ones provide the best perspectives into the underlying information.

When you place elements on the same report page, Power BI automatically ties them together. In this way, if you select an element in one visualization, the other visualizations will reflect the selection. For example, if you click the *Failed* bar for *User03* in the top right clustered bar chart, the corresponding data elements in the other visualizations will be selected, resulting in the nonrelated elements being grayed out.

Adding a Gauge to Your Report

Power BI Desktop also lets you add elements such as gauges, cards, and key performance indicators (KPIs) to a report. For example, I added the gauge shown in Figure 8 to display the number of `ROLLBACK` statements that were executed in response to the users' failed attempts to run T-SQL statements.

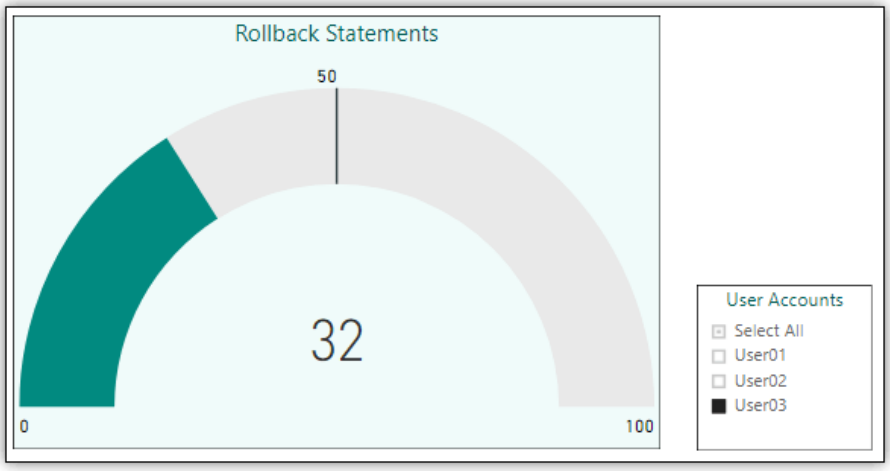


Figure 8. Adding a gauge to a report page

I also added a slicer for the user accounts to make it easy to see whether individual users are hitting the specified threshold. In Figure 8, the `User03` account is selected in the slicer, so the gauge shows only the number of rollbacks that pertain to that user. The gauge also specifies a target of *50,* but you can specify any target value.

The reason the target value is significant is that you can set alerts based on that value in the Power BI service (something you cannot do in Power BI Desktop). To use this feature, however, you must have a Power BI Pro license, which allows you to set up alerts that notify you regularly about potential issues. You can also set up alerts that generate email notifications on a regular basis. Plus, you can add alerts to card and KPI visuals.

Visualizing Audit Data in Power BI

Using Power BI to visualize SQL Server Audit data can provide you will a powerful tool for reviewing the logged information in a quick and efficient manner. What I've covered here only scratches the surface. There's far more to SQL Server Audit and Power BI Desktop, and both topics deserve much more attention.

But this article should at least offer you a good starting point for understanding what it takes to use Power BI to work with SQL Server Audit data. It's up to you to decide how far you want to take this and how extensively you might want to use the Power BI service, in addition to Power BI Desktop, keeping in mind the importance of protecting the audit data wherever it resides.