# MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

## DEPARTMENT OF BUSINESS MANAGEMENT

| | |
|---|---|
| **Course Name** | : **R22MBAB4 DATA ANALYSIS USING R AND TABLEAU** |
| **Academic year** | : 2023-24 |
| **Prescribed Textbook** | : Ben Jones, "Communicating Data with Tableau: Designing, Developing, and Delivering Data Visualizations", Shroff/O'Reilly |
| **Nature of the Course** | : BA Specialization Paper |

## PREFACE

**Course Aim:** R Studio is an integrated development environment (IDE) for the R programming language. It serves as a powerful tool for statistical analysis, data visualization, and data manipulation. Tableau is a powerful data visualization tool that enables professionals to create interactive and insightful dashboards.

**Learning Objective**: R Studio is a powerful tool for statistical analysis, offering advanced capabilities for hypothesis testing, regression modeling, and analyzing time-series data. It excels in data visualization, allowing the creation of detailed plots, charts, and graphs that reveal underlying data patterns and relationships. With robust data manipulation functions, R Studio simplifies the process of cleaning, transforming, and wrangling data. It also supports model building with its access to a wide range of machine learning algorithms. Moreover, its integration with Tableau enables a seamless enhancement of data analysis within Tableau's sophisticated visualizations. With hands-on applications and explanations and successfully outlines the necessary tools to make smart and successful business decisions.

## Unit-1: Introduction to the R language

**Introduction to the R language R Studio**: Introduction - Obtaining and Managing R - R Data Types and Objects, Classes, Creating and Accessing Objects - Data Structures in R - R Programming Fundamentals - Arithmetic and Matrix Operations - Advantages and Disadvantages of using R

| Objective | Outcome |
| --- | --- |
| It aims to equip learners with the knowledge to obtain and manage R packages, understand R's data types, objects, and classes, and to create and access these objects effectively. The chapter also delves into the various data structures in R, covers the fundamentals of R programming, and explains arithmetic and matrix operations | By the end of the chapter, readers should be able to recognize the advantages and disadvantages of using R, enabling them to utilize R efficiently for data analysis and statistical modeling. |
| **Overview** | |
| This unit provide complete installation and usage process and also explain the significant impact of subject in various fields, how the R support under different situations for decision making. | |

## Unit-2: Working with R

**Reading and Writing Data** - R Libraries - Functions and R Programming –The If Statement. Looping: for, repeat, while - Writing Functions - Function Arguments and Options - Basic R Commands. Graphics: Basic Plotting - Manipulating the Plotting Window - Advanced Plotting using Lattice Library - Saving Plots.

| Objective | Outcome |
| --- | --- |
| The objective of learning these R programming concepts is to equip individuals with the skills to perform data analysis and visualization efficiently. By understanding how to read and write data, utilize R libraries, and apply functions, one can manipulate datasets and execute conditional operations using the If statement. Mastery of looping constructs like for, repeat, and while loops is crucial for automating repetitive tasks. Additionally, gaining proficiency in writing functions and managing function arguments enhances code modularity and reusability. | The learning outcome includes the ability to create informative graphics, customize the plotting window, and employ advanced plotting techniques with the Lattice library, culminating in the capability to save and present data visualizations effectively. |
| **Overview** | |
| To develop a thorough skill set in R for comprehensive data analysis and effective visual communication of data insights. | |

## Unit-3 Standard Statistical Models in R & Statistics with R

**Models in R**: Standard Statistical Models in R - Model Formulae and Model Options - Output and Extraction from Fitted Models - Models Considered: Linear Regression: lm, Logistic regression: glm, Linear mixed models: lme.

**Statistics with R**: Summarizing Data - Calculating Relative Frequencies - Tabulating Factors and Creating Contingency Tables - Testing Categorical Variables for Independence - Calculating Quantiles of a Dataset - Converting Data into z-scores - t-test - testing Sample Proportions - Testing Normality - Comparing Means of Two Samples - Testing Correlation for Significance.

| Objective | Outcome |
| --- | --- |
| To understand and apply standard statistical models using R for data analysis, including linear regression, logistic regression, and linear mixed models, and to master statistical methods for summarizing and testing datasets. | Upon completion, the learner will be able to construct and interpret model formulae, extract meaningful insights from fitted models, and proficiently perform a variety of statistical tests to analyze and draw conclusions from data. |
| **Overview** | |
| This chapter provides a comprehensive overview of statistical modeling and analysis using R. It covers the creation and interpretation of standard statistical models, including linear regression with `lm`, logistic regression with `glm`, and linear mixed models with `lme`. Additionally, it delves into descriptive statistics techniques such as summarizing data, calculating relative frequencies, and creating contingency tables. The chapter also guides through inferential statistics methods, including hypothesis testing with t-tests, sample proportion tests, normality tests, and significance testing of correlations, equipping readers with the necessary tools to analyze and interpret their data effectively. | |

## Unit-4 : Introduction to Tableau

Tableau: Introduction - Terminology - Tableau User Interface - Basic Tableau Design Flow - Basic Visualization Design - Show Me! Choosing Mark Types Color - Size, and Shape Options - Shaped Axis Charts - Combination Charts - Measure Names - Measure Values - Data Connection - Connecting to Various Data Sources - Customizing Your View of the Datasets.

| Objective | Outcome |
| --- | --- |
| The objective of the content is to provide a comprehensive introduction to Tableau, a powerful data visualization tool. It aims to familiarize users with the essential terminology, the user interface, and the fundamental design flow for creating basic to advanced visualizations. | The outcome is that users will be able to connect to various data sources, customize their data views, and effectively use Tableau's features such as 'Show Me', mark types, and combination charts to transform raw data into meaningful insights. |
| **Overview** | |

Provides a comprehensive introduction to the world of data visualization with Tableau. It starts with the basics, familiarizing you with the terminology and the user interface. You'll learn the design flow and how to create basic visualizations. The syllabus covers the use of 'Show Me!' for choosing mark types and the application of color, size, and shape to convey information effectively. You'll explore various chart types, including shaped axis and combination charts, and delve into the nuances of measure names and values. The course also includes a crucial component on data connection, teaching you how to connect to different data sources and customize your dataset views for insightful analysis.

## Unit-5: Groups & Hierarchies in Datasets

Groups & Hierarchies: Groups - Hierarchies - Extracting Data - Data Blending - Charts - Bar Chart, Line Chart - Area Chart - Text Table/Cross Tab - Scatter Plot/Bubble Chart - Bullet Chart - Box Plot - Tree Map - Pie Chart - World Cloud - Tableau Maps - Geocoded Fields - Dashboard Actions - Distributing and Sharing Your Dashboards - Exporting Worksheets and Dashboards Publishing to Tableau Server - Creating Tableau Server User Filters.

| Objective | Outcome |
|---|---|
| The objectives of this course are to equip students with the skills to effectively organize data into groups and hierarchies, extract and blend data from various sources, and create a wide range of visualizations including bar charts, line charts, and more complex graphics like tree maps and scatter plots | The outcome for students will be the ability to build interactive dashboards, understand geocoded fields for mapping, and implement dashboard actions for a dynamic user experience. Additionally, students will learn to distribute, share, and publish their work on Tableau Server, ensuring a well-rounded mastery of Tableau's capabilities for data analysis and presentation. |

**Overview**

Encompasses a comprehensive overview of data visualization and dashboard creation using Tableau. It covers the foundational elements of grouping and hierarchy in data, methods for extracting and blending data from various sources, and a wide array of chart types including bar, line, and area charts. Additionally, it delves into more complex visualizations like scatter plots, bubble charts, and tree maps. The overview also includes practical aspects of Tableau such as geocoding, dashboard interactivity, distribution, and publishing, ensuring a well-rounded mastery of Tableau's capabilities for effective data analysis and storytelling.


**Dr. Kameswari Jada**
**Specialization- Business Analytics**

**Dr.G. Naveen Kumar**
Head of Department

**Dr. S. Srinivasa Rao**
Principal

## Introduction to R Programming Language

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool. It was designed by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team. R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. Moreover, the project conceives in 1992, with an initial version released in 1995 and a stable beta version in 2000.

### Why R Programming Language?

1. R programming is used as a leading tool for machine learning, statistics, and data analysis. Objects, functions, and packages can easily be created by R.
2. It's a platform-independent language. This means it can be applied to all operating system.
3. It's an open-source free language. That means anyone can install it in any organization without purchasing a license.
4. R programming language is not only a statistic package but also allows us to integrate with other languages (C, C++). Thus, you can easily interact with many data sources and statistical packages.

### Features of R Programming Language

**a. Statistical Features of R:**

1. Basic Statistics: The most common basic statistics terms are the mean, mode, and median. These are all known as "Measures of Central Tendency." So using the R language we can measure central tendency very easily.
2. Static graphics: R is rich with facilities for creating and developing interesting static graphics. R contains functionality for many plot types including graphic maps, mosaic plots, biplots, and the list goes on.
3. Probability distributions: Probability distributions play a vital role in statistics and by using R we can easily handle various types of probability distribution such as Binomial Distribution, Normal Distribution, Chi-squared Distribution and many more.
4. Data analysis: It provides a large, coherent and integrated collection of tools for data analysis.

**b. Programming Features of R:**

1. R Packages: One of the major features of R is it has a wide availability of libraries. R has CRAN(Comprehensive R Archive Network), which is a repository holding more than 10, 0000 packages.
2. Distributed Computing: Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency and performance. Two new packages ddR and multidplyr used for distributed programming in R were released in November 2015.

**c. Programming in R:**

Since R is much similar to other widely used languages syntactically, it is easier to code and learn in R. Programs can be written in R in any of the widely used IDE like R Studio, Rattle, Tinn-R, etc. After writing the program save the file with the extension .r. To run the program use the following command on the command line: R file_name.r

### Advantages of R:

1. R is the most comprehensive statistical analysis package. As new technology and concepts often appear first in R.
2. As R programming language is an open source. Thus, you can run R anywhere and at any time.
3. R programming language is suitable for GNU/Linux and Windows operating system.
4. R programming is cross-platform which runs on any operating system.
5. In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.

### Disadvantages of R:

1. In the R programming language, the standard of some packages is less than perfect.
2. Although, R commands give little pressure to memory management. So R programming language may consume all available memory.
3. In R basically, nobody to complain if something doesn't work.
4. R programming language is much slower than other programming languages such as Python and MATLAB.

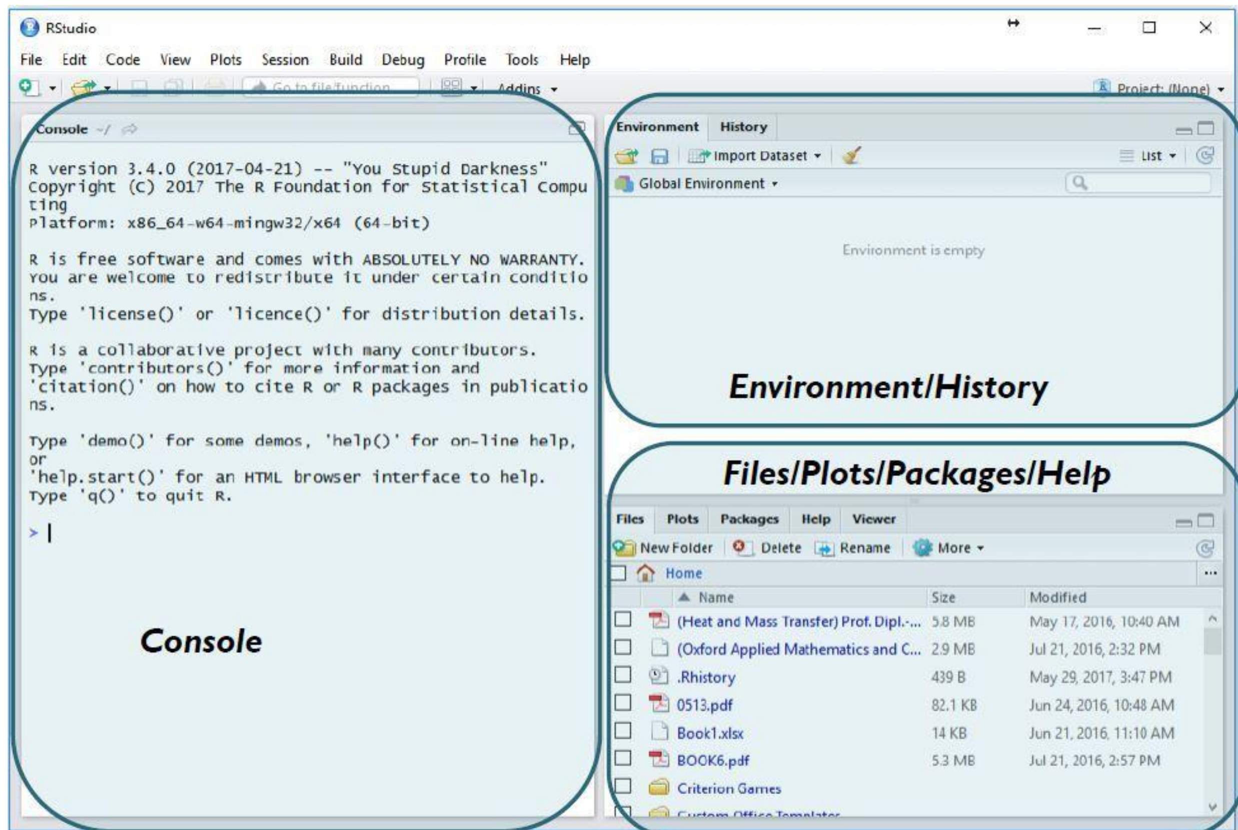
## Introduction to R Studio

R Studio is an integrated development environment (IDE) for R. IDE is a GUI, where you can write your quotes, see the results and also see the variables that are generated during the course of programming.

- R Studio is available as both Open source and Commercial software.
- R Studio is also available as both Desktop and Server versions.
- R Studio is also available for various platforms such as Windows, Linux, and macOS.

Rstudio is an open-source tool that provides Ide to use R language, and enterprise-ready professional software for data science teams to develop share the work with their team. R Studio can be downloaded from its official Website (https://rstudio.com

After the installation process is over, the R Studio interface looks like:




- The console panel(left panel) is the place where R is waiting for you to tell it what to do, and see the results that are generated when you type in the commands.
- To the top right, you have the Environmental/History panel. It contains 2 tabs:
  - Environment tab: It shows the variables that are generated during the course of programming in a workspace that is temporary.
  - History tab: In this tab, you'll see all the commands that are used till now from the start of usage of R Studio.

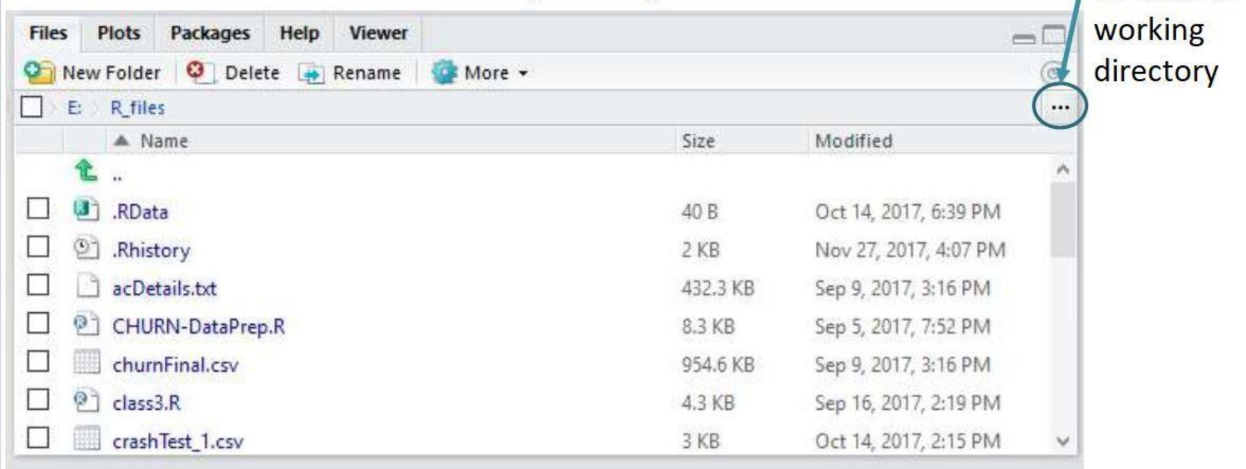
1. To the right bottom, you have another panel, which contains multiple tabs, such as files, plots, packages, help, and viewer.
   - The Files tab shows the files and directories that are available within the default workspace of R.
   - The Plots tab shows the plots that are generated during the course of programming.
   - The Packages tab helps you to look at what are the packages that are already installed in the R Studio and it also gives a user interface to install new packages.
   - The Help tab is the most important one where you can get help from the R Documentation on the functions that are in built-in R.
   - The final and last tab is that the Viewer tab which can be used to see the local web content that's generated using R.

## Set the working directory in R Studio

R is always pointed at a directory on our computer. We can find out which directory by running the getwd() function. Note: this function has no arguments. We can set the working directory manually in two ways:

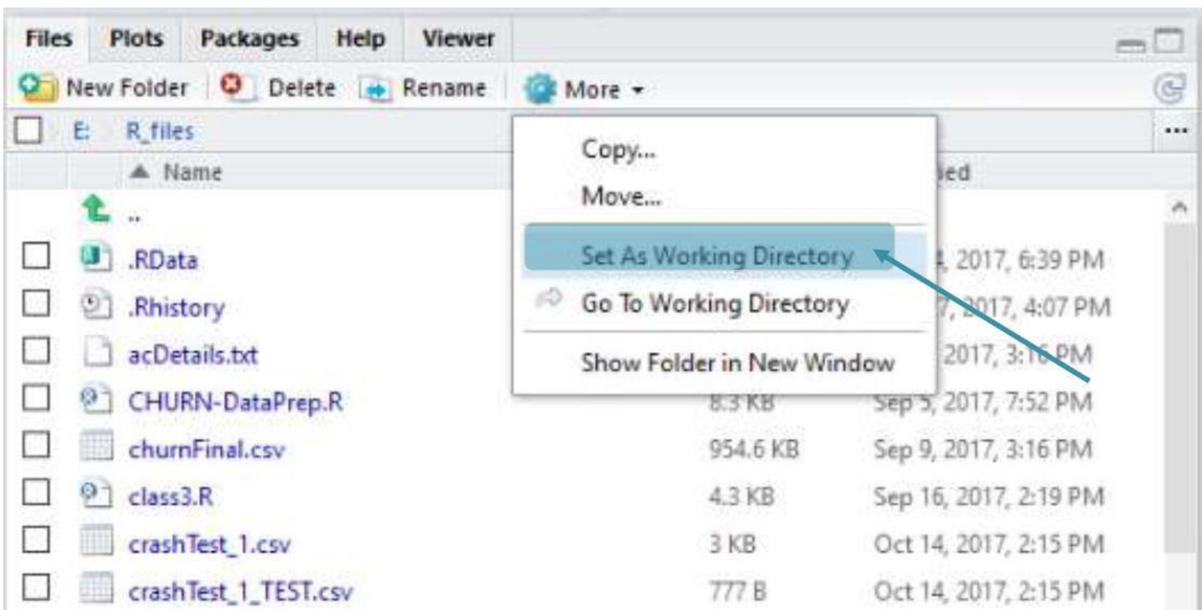
- The first way is to use the console and using the command setwd("directorypath").
- You can use this function setwd() and give the path of the directory which you want to be the working directory for R studio, in the double codes.

- The second way is to set the working directory from the GUI.
- To set the working directory from the GUI you have to click on this 3 dots button. When you click this, this will open up a file browser, which will help you to choose your working directory.




- Once you choose your working directory, you need to use this setting button in the more tab and click it and then you get a popup menu, where you need to select "Set as working directory".

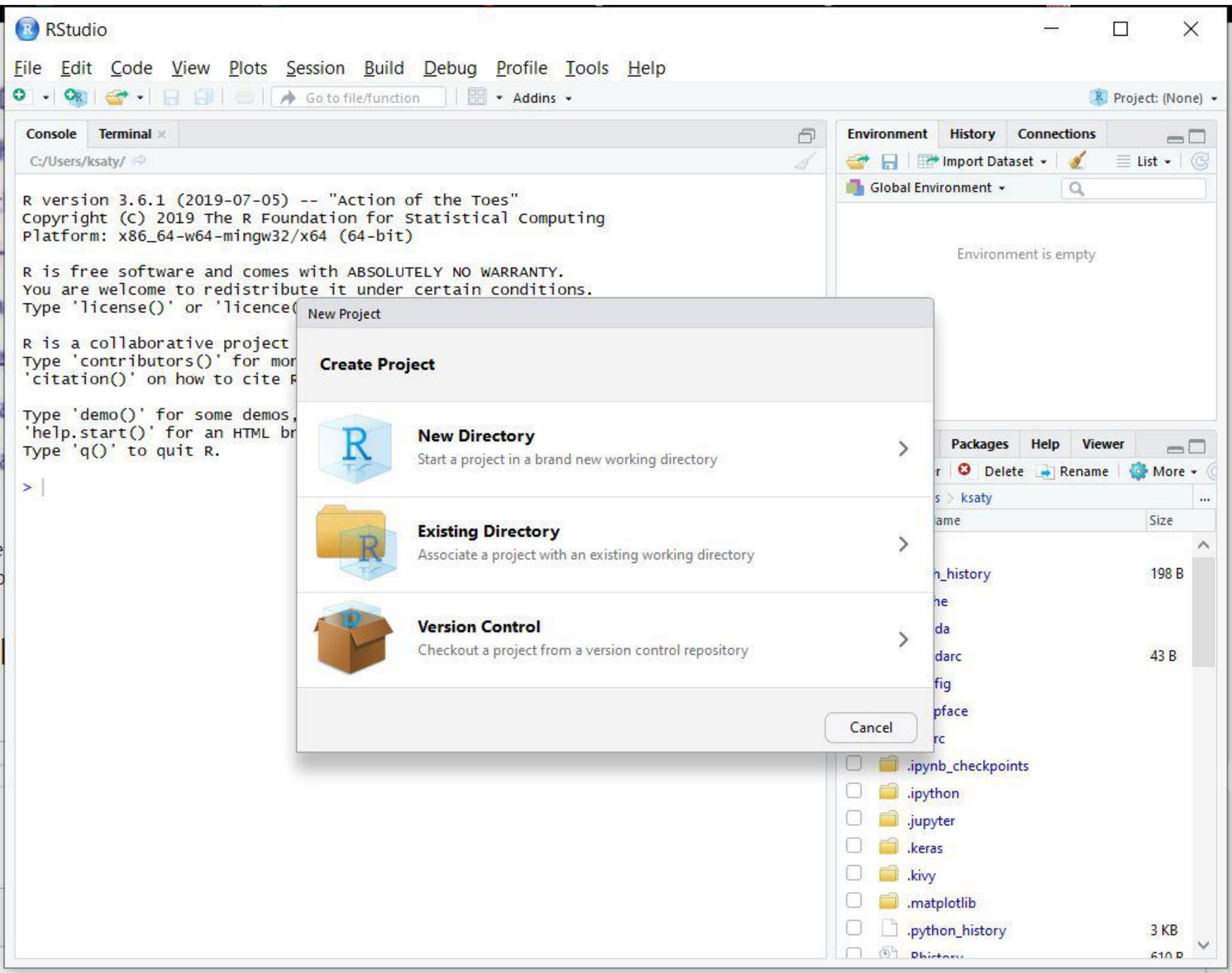
***STEP 2: Once directory is chosen, select the more icon and choose "Set as Working Directory"***



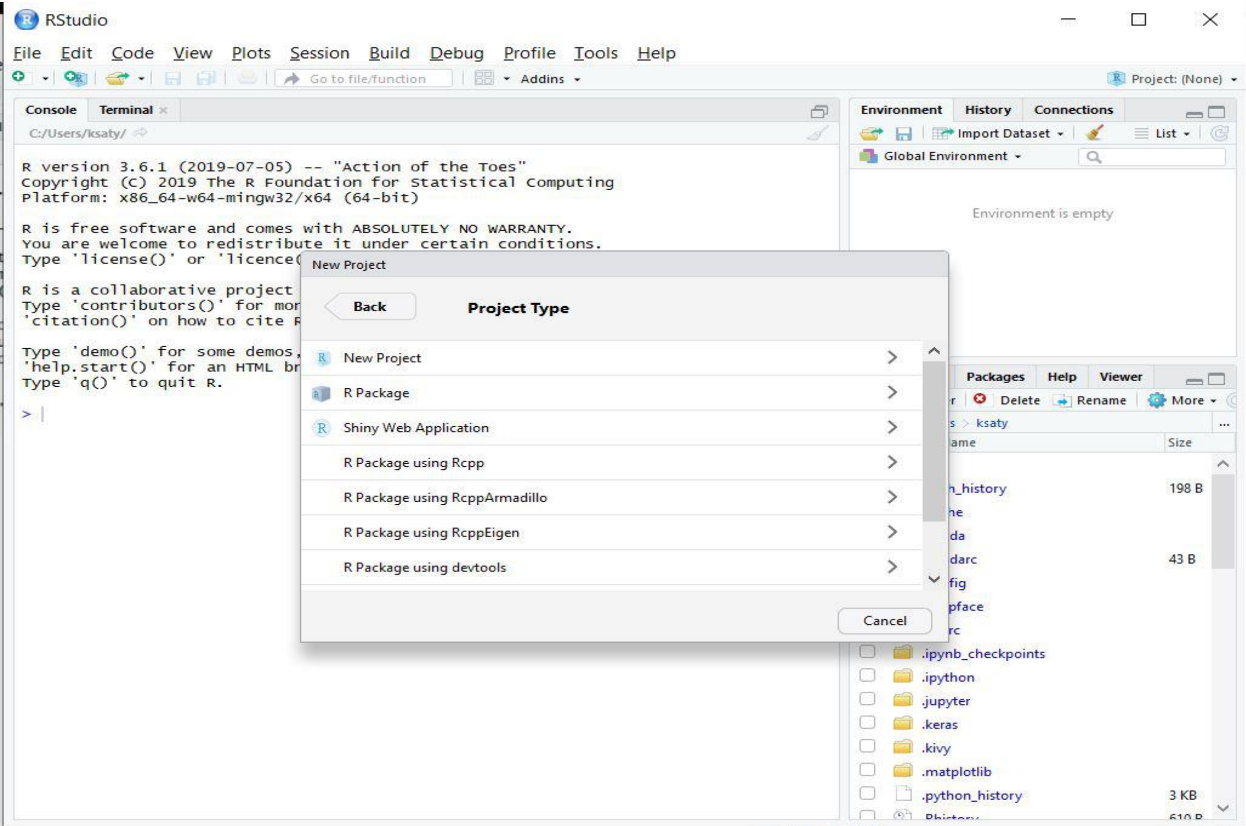

This will select the current directory, which you have chosen using this file browser as your working directory. Once you set the working directory, you are ready to program in R Studio.

### Create an RStudio project

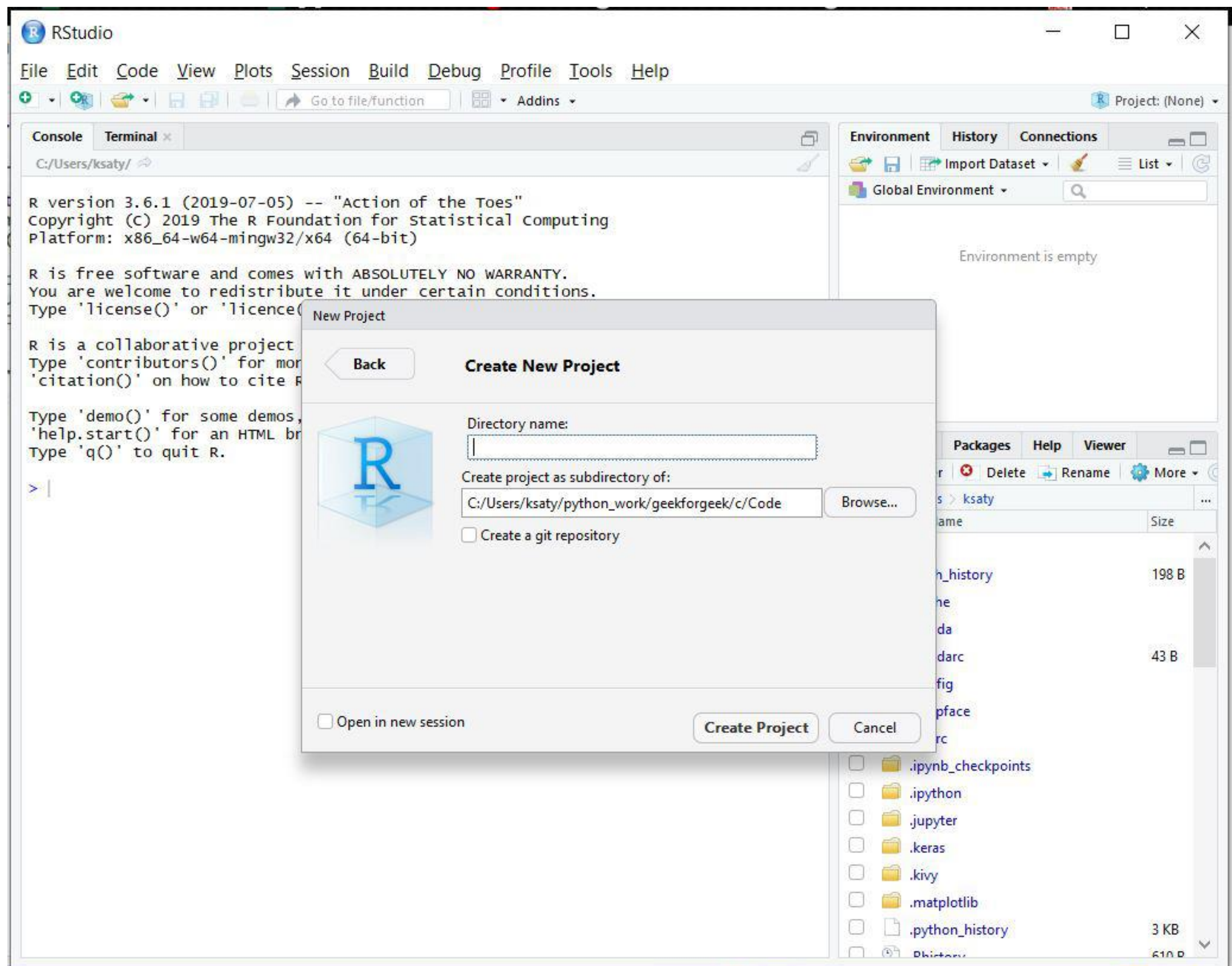
1. Select the FILE option and select create option.



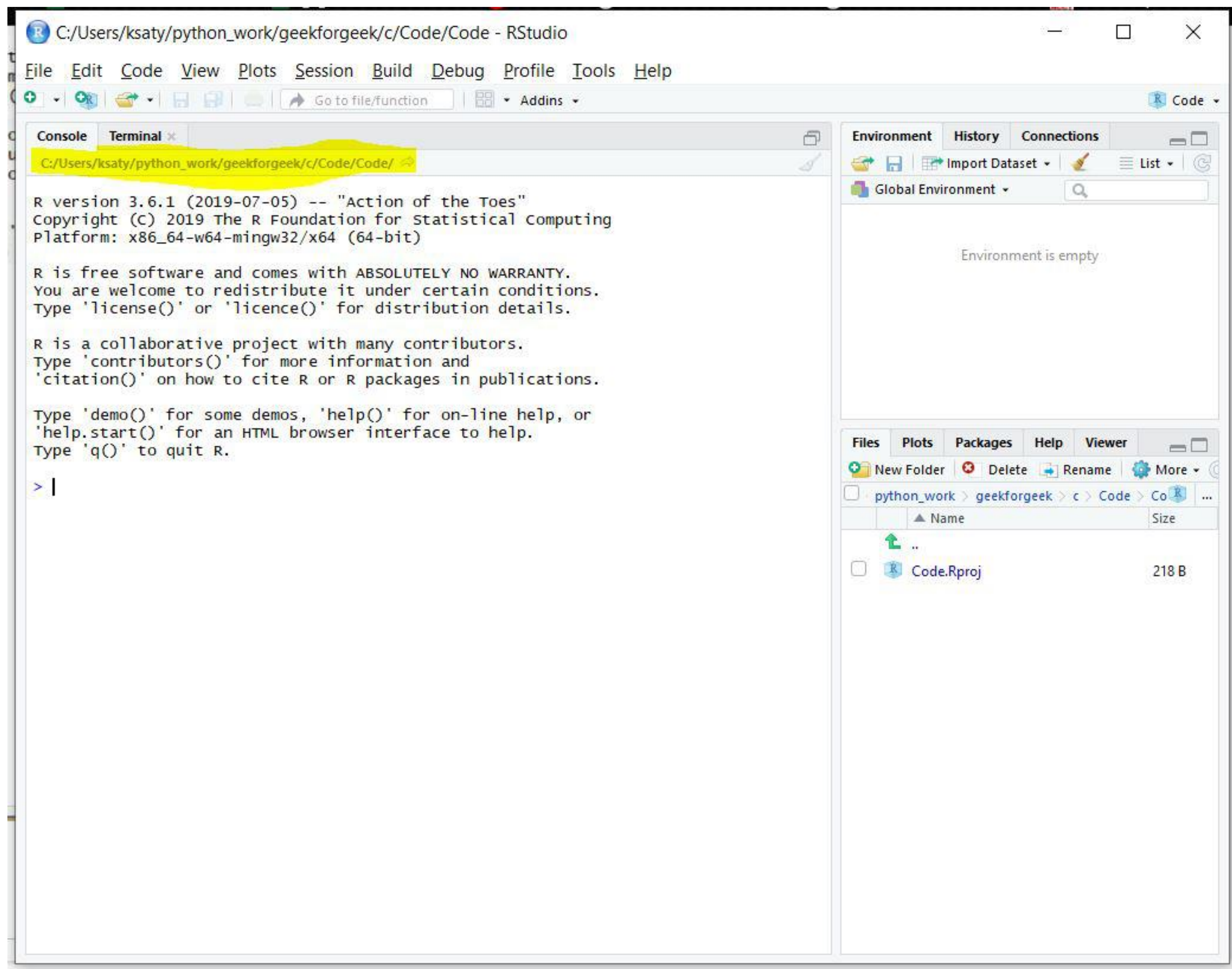

2. Then select the New Project option.




3. Then choose the path and directory name.




4. Finally, project are created in a specific location:



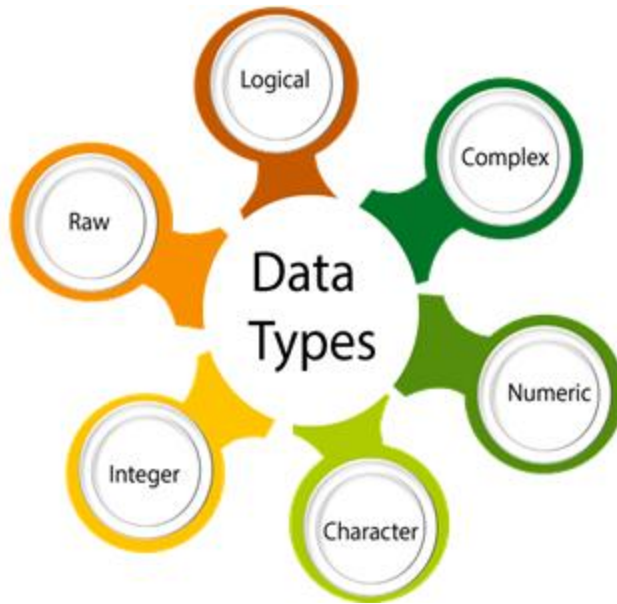

## Navigating directories in R studio

- getwd(): Returns the current working directory.
- setwd(): Set the working directory.
- dir(): Return the list of the directory.

## Data Types in R Programming

In R, there are several data types such as integer, string, etc. The operating system allocates memory based on the data type of the variable and decides what can be stored in the reserved memory. Each variable in R has an associated data type and has some specific operations that can be performed over it.

There are the following data types which are used in R programming:




The following table shows the data type and the values that each data type can take.

| Data type | Example | Description |
|---|---|---|
| **Logical** | True, False | It is a special data type for data with only two possible values which can be construed as true/false. |
| **Numeric** | 12,32,112,5432 | Decimal value is called numeric in R, and it is the default computational data type. |
| **Integer** | 3L, 66L, 2346L | Here, L tells R to store the value as an integer, |
| **Complex** | Z=1+2i, t=7+3i | A complex value in R is defined as the pure imaginary value i. |
| **Character** | 'a', '"good"', "TRUE" '35.4' | In R programming, a character is used to represent string values. We convert objects into character values with the help ofas.character() function. |
| **Raw** | | A raw data type is used to holds raw bytes. |

### Numeric Datatype

Decimal values are called numeric in R. It is the default data type for numbers in R. If you assign a decimal value to a variable x as follows, x will be of numeric type.

```
# Assign a decimal value to x
x = 5.6
# print the class name of variable
print(class(x))
Output:
[1] "numeric"
```

Even if an integer is assigned to a variable y, it is still being saved as a numeric value.

```
# Assign an integer value to y
y = 5
 # print the class name of variable
print(class(y))
```

Output:
[1] "numeric"

### Logical Datatype

R has logical data types that take either a value of true or false. A logical value is often created via a comparison between variables.

```
# Sample values
x = 4
y = 3
 # Comparing two values
z = x > y
 # print the logical value
print(z)
# print the class name of z
print(class(z))
```

Output:
[1] TRUE
[1] "logical"

### Complex Datatype

R supports complex data types that are set of all the complex numbers. The complex data type is to store numbers with an imaginary component.

```
# Assign a complex value to x
x = 4 + 3i
 # print the class name of x
print(class(x))
```

Output:
[1] "complex"

### Character Datatype

R supports character data types where you have all the alphabets and special characters. It stores character values or strings. Strings in R can contain alphabets, numbers, and symbols. The easiest way to denote that a value is of character type in R is to wrap the value inside single or double inverted commas.

```
# Assign a character value to char
char = "Geeksforgeeks"
 # print the class name of char
print(class(char))
```

Output:
[1] "character"

## R Classes and Objects

We can do object oriented programming in R. In fact, everything in R is an object. An object is a data structure having some attributes and methods which act on its attributes. Class is a blueprint for the object. An object is also called an instance of a class and the process of creating this object is called instantiation. While most programming languages have a single class system, R has three class systems. Namely, S3, S4 and more recently Reference class systems (e.g. R6).

S3:-It is being used to overload any function. It means calling a different name of the function and It depends upon the type of input parameter or the number of a parameter.

S4:-It is the characteristic OOP system, but it is tricky to debug. Reference classes are the modern alternative for S4 classes.

Classes are used as the outline or design for the object. It encapsulates the data members along with the functions

In R, all types of data are treated as objects. However, objects are not simply collections of data. They are particular instances (instantiations) of particular classes. Operations, or functions, are defined for specific classes. The variables are assigned with R-Objects and the data type of the R-object becomes the data type of the variable. There are many types of R-objects. The frequently used ones are:

1. Vectors
2. Lists
3. Matrices
4. Arrays
5. Factors
6. Data Frames.

## Creating objects in R

To create an object, we need to give it a name followed by the assignment operator <- or an equal sign = and the value we want to give it. For example, the following command assigns the value 5 to the object x (x is object name)

x <- 5 (left assignment)

5-> x (right assignment)

x=5

After this assignment, the object x ‘contains’ the value 5.

Another assignment to the same object will change the content.

x = 107

You can check the content of an object by simply entering the name of the object on an interactive command line.

## Data Structures in R Programming

A data structure is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. Data structures in R programming are tools for holding multiple values.

R’s base data structures are often organized by their dimensionality (1D, 2D, or nD) and whether they’re homogeneous (all elements must be of the identical type) or heterogeneous (the elements are often of various types). This gives rise to the six data types which are most frequently utilized in data analysis.

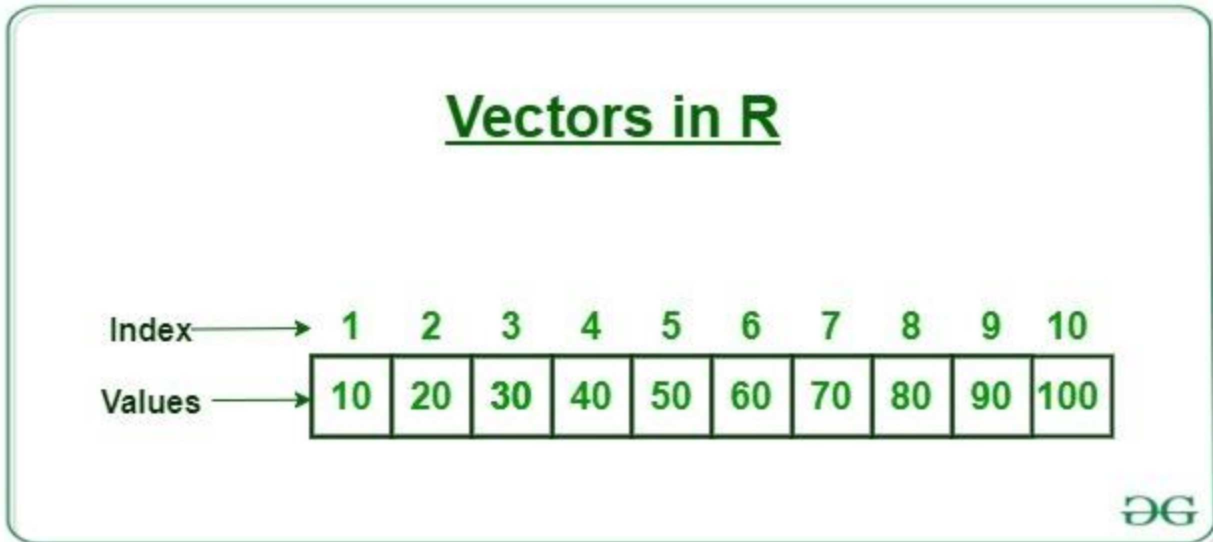
The most essential data structures used in R include:

1. Vectors
2. Lists

3. Dataframes
4. Matrices
5. Arrays
6. Factors

**1. Vectors**

A vector is the basic data structure in R, or we can say vectors are the most basic R data objects. There are six types of atomic vectors such as logical, integer, character, double, and raw. "A vector is a collection of elements which is most commonly of mode character, integer, logical or numeric"




### Numeric vectors

Numeric vectors are those which contain numeric values such as integer, float, etc. # Vectors(ordered collection of same data type)

```
x = c(1, 3, 5, 7, 8)
# Printing those elements in console
print(x)
# display type of vector
typeof(v1)
Output:
[1] 1 3 5 7 8
[1] "integer"
```

### Character vectors

Character vectors contain alphanumeric values and special characters.

```
v1 <- c('geeks', '2', 'hello', 57)
# Displaying type of vector
typeof(v1)
Output:
[1] "character"
```

### Logical vectors

Logical vectors contain boolean values such as TRUE, FALSE and NA for Null values.

```
# using c() function
v1 <- c(TRUE, FALSE, TRUE, NA)
# Displaying type of vector
typeof(v1)
Output:
[1] "logical"
```

2. **List**

In R, the list is the container. Unlike an atomic vector, the list is not restricted to be a single mode. A list contains a mixture of data types. The list is also known as generic vectors because the element of the list can be of any type of R object. "A list is a special type of vector in which each element can be a different type."

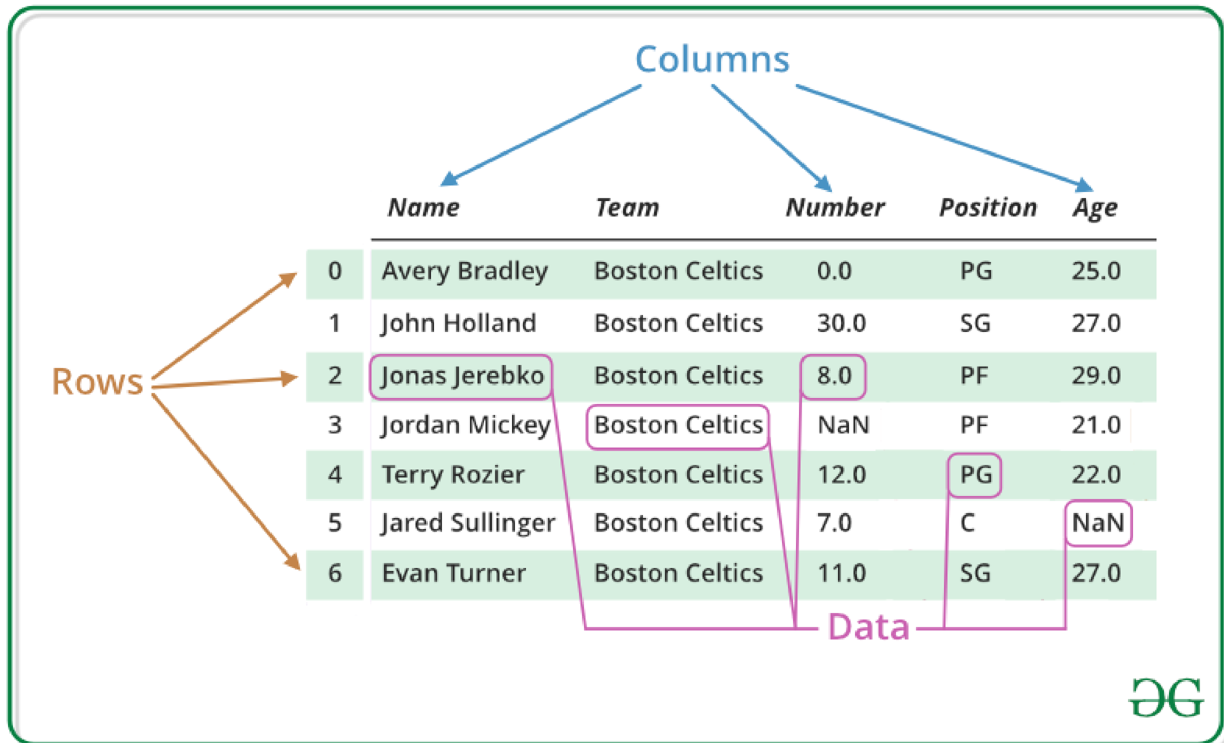
A list in R is a generic object consisting of an ordered collection of objects. Lists are one-dimensional, heterogeneous data structures. The list can be a list of vectors, a list of matrices, a list of characters and a list of functions, and so on.

A list is a vector but with heterogeneous data elements. A list in R is created with the use of list() function. R allows accessing elements of a list with the use of the index value. In R, the indexing of a list starts with 1 instead of 0 like other programming languages. To create a List in R you need to use the function called “list()”.

```
empList = list(("Debi", "Sandeep",1, 2, 3)
typeof(empList)
Output:
[1] “list”
```

3. **Dataframes**

Data Frames in R Language are generic data objects of R which are used to store the tabular data. Data frames can also be interpreted as matrices where each column of a matrix can be of the different data types. DataFrame is made up of three principal components, the data, rows, and columns.




**Characteristics of a data frame:**

1. The column name will be non-empty.
2. The row names will be unique.
3. A data frame stored numeric, factor or character type data.
4. Each column will contain same number of data items.

### Creating a Dataframe

To create a data frame in R use data.frame() command and then pass each of the vectors you have created as arguments to the function.

```
# creating a data frame
friend.data <- data.frame(
   friend_id = c(1:5),
   friend_name = c("Sachin", "Sourav",
                   "Dravid", "Sehwag",
                   "Dhoni"),
   stringsAsFactors = FALSE
)
# print the data frame
print(friend.data)
```

**4. Matrices**

A matrix is an R object in which the elements are arranged in a two-dimensional rectangular layout. In the matrix, elements of the same atomic types are contained. For mathematical calculation, this can use a matrix containing the numeric element. A matrix is created with the help of the matrix() function in R. Matrix is a rectangular arrangement of numbers in rows and columns.

**Creating a matrix**

To create a matrix in R you need to use the function called matrix(). The arguments to this matrix() are the set of elements in the vector. You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix.

```
# R program to create a matrix
A = matrix(
# Taking sequence of elements
c(1, 2, 3, 4, 5, 6, 7, 8, 9),
# No of rows
nrow = 3,
# No of columns
ncol = 3,
# By default matrices are in column-wise order
# So this parameter decides how to arrange the matrix
byrow = TRUE)
# Naming rows
rownames(A) = c("a", "b", "c")
# Naming columns
colnames(A) = c("c", "d", "e")
cat("The 3x3 matrix:\n")
print(A)
```

Output:

```
 The 3x3 matrix:
  c d e
a 1 2 3
b 4 5 6
c 7 8 9
```

5. **Arrays**

Arrays are essential data storage structures defined by a fixed number of dimensions. Arrays are used for the allocation of space at contiguous memory locations. Uni-dimensional arrays are called vectors with the length being their only dimension. Two-dimensional arrays are called matrices, consisting of fixed numbers of rows and columns. Arrays consist of all elements of the same data type. Vectors are supplied as input to the function and then create an array based on the number of dimensions.

There is another type of data objects which can store data in more than two dimensions known as arrays. "An array is a collection of a similar data type with contiguous memory allocation." Suppose, if we create an array of dimension (2, 3, 4) then it creates four rectangular matrices of two rows and three columns.

In R, an array is created with the help of array() function. This function takes a vector as an input and uses the value in the dim parameter to create an array.

**Creating an Array**

An array in R can be created with the use of array() function. List of elements is passed to the array() functions along with the dimensions as required.

```
arr = array(2:13, dim = c(2, 3, 2))
```

```
print(arr)
```

6. **Factors**

Factors are also data objects that are used to categorize the data and store it as levels. Factors can store both strings and integers. Columns have a limited number of unique values so that factors are very useful in columns. It is very useful in data analysis for statistical modeling.
Factors in R Programming Language are data structures that are implemented to categorize the data or represent categorical data and store it on multiple levels.
They can be stored as integers with a corresponding label to every unique integer. Though factors may look similar to character vectors, they are integers and care must be taken while using them as strings. The factor accepts only a restricted number of distinct values.
Factors are created with the help of factor() function by taking a vector as an input parameter.

**Creating a Factor**

The command used to create or modify a factor in R language is – factor() with a vector as input.
The two steps to creating a factor are:

1. Creating a vector
2. Converting the vector created into a factor using function factor()

```
# Creating a vector
x < -c("female", "male", "male", "female")
print(x)
# Converting the vector x into a factor
# named gender
gender < -factor(x)
print(gender)
```

R Programming Fundamentals

A program in R is made up of three things: Variables, Comments, and Keywords. Variables are used to store the data, Comments are used to improve code readability, and Keywords are reserved words that hold a specific meaning to the compiler.

### Variables in R

Variables which like any other programming language are the name given to reserved memory locations that can store any type of data.

```
var1 = "MRCET"
var2 <- 105
```

### Comments in R

Comments are a way to improve your code's readability and are only meant for the user so the interpreter ignores it. Only single-line comments are available in R but we can also use multiline comments by using a simple trick which is shown below. Single line comments can be written by using # at the beginning of the statement.

*# R is the most popular language used for Statistical Computing and Data Analysis*

### Keywords in R

Keywords are the words reserved by a program because they have a special meaning thus a keyword can't be used as a variable name, function name, etc. We can view these keywords by using either
help(reserved)
or
?reserved.

## R – Operators

Operators are the symbols directing the compiler to perform various kinds of operations between the operands. Operators simulate the various mathematical, logical, and decision operations performed on a set of Complex Numbers, Integers, and Numericals as input operands.

**Types of the operators**

1. Arithmetic Operators
2. Logical Operators
3. Relational Operators
4. Assignment Operators
5. Miscellaneous Operator

### Arithmetic Operators

Arithmetic operations simulate various math operations, like addition, subtraction, multiplication, division, and modulo using the specified operator between operands, which may be either scalar values, complex numbers, or vectors. The operations are performed element-wise at the corresponding positions of the vectors.

Addition operator (+):

The values at the corresponding positions of both the operands are added.

```
Input : a <- c (1, 0.1)
        b <- c (2.33, 4)
        print (a+b)
Output : 3.33  4.10
```

Subtraction Operator (-):

The second operand values are subtracted from the first. Consider the following R snippet to subtract two variables:

```
Input : a <- 6
        b <- 8.4
        print (a-b)
Output : -2.4
```

Multiplication Operator (*):

The multiplication of corresponding elements of vectors and Integers are multiplied with the use of '*' operator.

```
Input : B= matrix(c(4,6i),nrow=1,ncol=2)
        C= matrix(c(2,2i ),nrow=1, ncol=2)
        print (B*C)
Output : 8+0i  -12+0i
The elements at corresponding positions of matrices are multiplied.
```

Division Operator (/):

The first operand is divided by the second operand with the use of '/' operator.

```
Input : a <- 1
        b <- 0
        print (a/b)
Output : -Inf
```

Power Operator (^):
The first operand is raised to the power of the second operand.

```
Input : list1 <- c(2, 3)
        list2 <- c(2,4)
        print(list1^list2)
Output : 4  81
```

Modulo Operator (%%):
The remainder of the first operand divided by the second operand is returned.

```
Input : list1<- c(2, 3)
        list2<-c(2,4)
        print(list1%%list2)
Output : 0  3
```

### Logical Operators

Logical operations simulate element-wise decision operations, based on the specified operator between the operands, which are then evaluated to either a True or False boolean value. Any non zero integer value is considered as a TRUE value, be it complex or real number.
**Element-wise Logical AND operator (&):** Returns True if both the operands are True.
**Element-wise Logical OR operator (|):** Returns True if either of the operands are True.
**NOT operator (!):** A unary operator that negates the status of the elements of the operand.
**Logical AND operator (&&):** Returns True if both the first elements of the operands are True.
**Logical OR operator (||):** Returns True if either of the first elements of the operands are True.

### Relational Operators

The relational operators carry out comparison operations between the corresponding elements of the operands. Returns a boolean TRUE value if the first operand satisfies the relation compared to the second. A TRUE value is always considered to be greater than the FALSE.
**Less than (<):**
Returns TRUE if the corresponding element of the first operand is less than that of the second operand. Else returns FALSE.
**Less than equal to (<=):**
Returns TRUE if the corresponding element of the first operand is less than or equal to that of the second operand. Else returns FALSE.
**Greater than (>):**
Returns TRUE if the corresponding element of the first operand is greater than that of the second operand. Else returns FALSE.
**Greater than equal to (>=):**

Returns TRUE if the corresponding element of the first operand is greater or equal to than that of the second operand. Else returns FALSE.

**Not equal to (!=):**

Returns TRUE if the corresponding element of the first operand is not equal to second operand. Else returns FALSE.

# CHAPTER II

## Reading and Writing Data

### Reading CSV Files

A popular data file format is the text file format where columns are separated by a tab, space or comma.

**Reading from a comma delimitted (CSV) file**

**read.csv()** function to read a CSV file available in your current working directory with the following command:

dat <- read.csv("your.path/filename.csv", header=TRUE)

Here you would substitute the location of your file (i.e. the file directory where it is stored), as well as the particular file name. Note that to reference a file in the path, you use either one forward slash (/) as above, or two backward slashes (\\).The <- is called an assignment operator.

1. Reading TXT files separated by TAB delimited

   The read.table() function can be used to import this file. The argument **sep='\t'** specifies a TAB as the variable delimiter

   **dat <- read.table(' your.path/filename.csv, header=T, sep='\t')**
2. Reading TXT files separated by Comma delimited (.CSV)

   **dat <- read.table(' your.path/filename.csv, header=T, sep=',')**
3. Reading TXT files separated by Blank space (.TXT)

   **dat <- read.table(' your.path/filename.csv, header=T, sep='')**

### Reading Excel Files

To load Excel files into R, first we need to run library "gdata"

dat <- read.xlsx(' your.path/filename.xlsx', sheet=1)

Using other library packages

To load Excel files into R, first we need to install the following

1. Install Java version (i.e. 64-bit Java or 32-bit Java) on the computer
2. Install "xlsx" package in R environment

   install.packages("xlsx")
3. Run following library packages in R

   library(rJava)

   library(xlsx)

Now we can import Excel files in R using read.xlsx() function

read.xlsx(file_name,sheetIndex)

dat <- read.xlsx(' **your.path/filename.xlsx', sheet=1)**

### Reading JSON file

To read "jSON" file install "rjson" package

install.packages("rjson")
Run the “rjson” library in R
library(rjson)
Now we can import JSONfile in R using fromJSON() function

```
dat <- fromJSON(file = "employee_info.json")
```

### Writing the CSV Files

R also allows us to write into the .csv file. For this purpose, R provides a write.csv() function. This function creates a CSV file from an existing data frame. This function creates the file in the current working directory.

```
write.csv(my_data, file = "my_data.csv")
write.csv2(my_data, file = "my_data.csv")
```

### Writing Data to text files

Writing to .txt files is very similar to that of the CSV files. Following is the syntax to write to a text file:
And uses “.” for the decimal point, a comma (“, ”) and a semicolon (“;”) for the separator.

```
write.table(my_data, file = "my_data.txt", sep = "")
```

### Writing Data to Excel files

To write data to excel we need to install the package known as “xlsx package”, it is basically a java based solution for reading, writing, and committing changes to excel files. It can be installed as follows:

```
install.packages("xlsx")
```

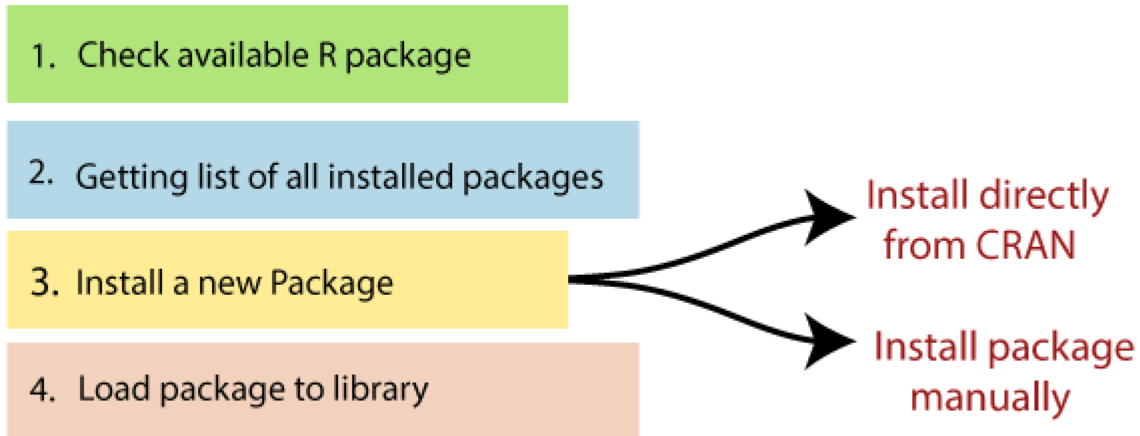
Write.xlsx() function is used to write to a excel file:

```
library("xlsx")

write.xlsx(my_data, file = "result.xlsx",
           sheetName = "my_data", append = FALSE).
```

## R Packages

R packages are the collection of R functions, sample data, and compile codes. In the R environment, these packages are stored under a directory called "library." During installation, R installs a set of packages. We can add packages later when they are needed for some specific purpose. Only the default packages will be available when we start the R console. Other packages which are already installed will be loaded explicitly to be used by the R program.
There is the following list of commands to be used to check, verify, and use the R packages




**Check Available R Packages**

R provides libPaths() function to find the library locations.

```
libPaths()
```

**Getting the list of all the packages installed**

R provides library() function, which allows us to get the list of all the installed packages.

```
library()
```

### Install a New Package

In R, there are two techniques to add new R packages. The first technique is installing package directly from the CRAN directory, and the second one is to install it manually after downloading the package to our local system.

```
install.packages("Package Name")
```

```
install.packages("XML")
```

### Load Package to Library

We cannot use the package in our code until it will not be loaded into the current R environment. We also need to load a package which is already installed previously but not available in the current environment.

```
library("package Name", lib.loc = "path to library")
```

### List of R packages

R is the language of data science which includes a vast repository of packages. These packages appeal to different regions which use R for their data purposes. CRAN has 10,000 packages, making it an ocean of superlative statistical work. There are some mostly used and popular packages which are as follows:

1. tidyr
   The word tidyr comes from the word tidy, which means clear. So the tidyr package is used to make the data' tidy'. This package works well with dplyr. This package is an evolution of the reshape2 package.
2. ggplot2

R allows us to create graphics declaratively. R provides the ggplot package for this purpose. This package is famous for its elegant and quality graphs which sets it apart from other visualization packages.

3. ggraph
   R provides an extension of ggplot known as ggraph. The limitation of ggplot is the dependency on tabular data is taken away in ggraph.
4. dplyr
   R allows us to perform data wrangling and data analysis. R provides the dplyr library for this purpose. This library facilitates several functions for the data frame in R.
5. caret
   R allows us to perform classification and regression tasks by providing the caret package. CaretEnsemble is a feature of caret which is used for the combination of different models.
6. MASS
   The MASS package provides a large number of statistical functions. It provides datasets that are in conjunction with the book "Modern Applied Statistics with S."
7. e1071
   The e1071 library provides useful functions which are essential for data analysis like Naive Bayes, Fourier Transforms, SVMs, Clustering, and other miscellaneous functions.

## Functions in R Programming

Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In R Programming Language when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more function definitions in a single R file.

A function is a set of statements orchestrated together to perform a specific operation. A function is an object so the interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions. The function in turn performs the task and returns control to the interpreter as well as any return values that may be stored in other objects.

### Types of function in R Language

1. **Built-in Function:** Built function R is sq(), mean(), max(), these function are directly call in the program by users.
2. **User-defined Function:** R language allow us to write our own function.

### Syntax:

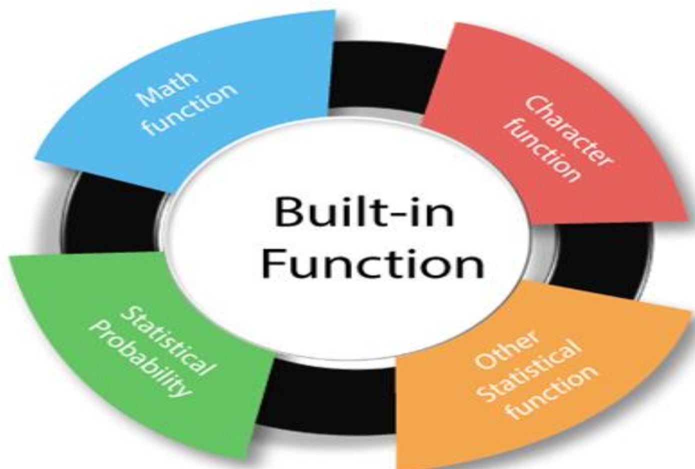
```
function_name = function(arg_1, arg_2, ...)
{
Function body
}
```

The various components/parts of a function are:

- **Function name:** It is the actual name of the function. It is stored in R environment as an object with this name.
- **Arguments:** An argument is a placeholder. Whenever a function is invoked, a value if passed to the argument. They are optional; that is, a function may contain no arguments. Also arguments can have default values.
- **Function Body:** It contains all the set of statements that defines what actually the function does.
- **Return Values:** It is the values that function returns after the successful execution of the tasks.In more general,it is the last expression in the function body to be evaluated.

### R Built-in Functions

The functions which are already created or defined in the programming framework are known as a built-in function. R has a rich set of functions that can be used to perform almost every task for the user. These built-in functions are divided into the following categories based on their functionality.




- **Math Functions**

  R provides the various mathematical functions to perform the mathematical calculation. These mathematical functions are very helpful to find absolute value, square value and much more calculations.

| S. No | Function | Description | Example |
|---|---|---|---|
| **1.** | abs(x) | It returns the absolute value of input x. | x<- -4<br>print(abs(x))<br>**Output**<br>[1]  4 |
| **2.** | sqrt(x) | It returns the square root of input x. | x<- 4<br>print(sqrt(x))<br>**Output**<br>[1]  2 |
| **3.** | ceiling(x) | It returns the smallest integer which is larger than or equa to x. | x<- 4.5<br>print(ceiling(x))<br>**Output**<br>[1]  5 |

| 4. | floor(x) | It returns the largest integer, which is smaller than or equal to x. | x<- 2.5<br>print(floor(x))<br>**Output**<br>[1] 2 |
|---|---|---|---|
| 5. | trunc(x) | It returns the truncate value of input x. | x<- c(1.2,2.5,8.1)<br>print(trunc(x))<br>**Output**<br>[1] 1 2 8 |
| 6. | round(x, digits=n) | It returns round value of input x. | x<- -4<br>print(abs(x))<br>**Output**<br>4 |
| 7. | cos(x), sin(x), tan(x) | It returns cos(x), sin(x) value of input x. | x<- 4<br>print(cos(x))<br>print(sin(x))<br>print(tan(x))<br>**Output**<br>[1] -06536436<br>[2] -0.7568025<br>[3] 1.157821 |
| 8. | log(x) | It returns natural logarithm of input x. | x<- 4<br>print(log(x))<br>**Output**<br>[1] 1.386294 |
| 9. | log10(x) | It returns common logarithm of input x. | x<- 4<br>print(log10(x))<br>**Output**<br>[1] 0.60206 |
| 10. | exp(x) | It returns exponent. | x<- 4<br>print(exp(x))<br>**Output**<br>[1] 54.59815 |

### String Function

R provides various string functions to perform tasks. These string functions allow us to extract sub string from string, search pattern etc. There are the following string functions in R:

| S. No | Function | Description | Example |
|---|---|---|---|
| 1. | substr(x, start=n1,stop=n2) | It is used to extract substrings in a character vector. | a <- "987654321"<br>substr(a, 3, 3) |

| | | | **Output**<br>[1] "3" |
|---|---|---|---|
| **2.** | grep(pattern, x ignore.case=FALSE, fixed=FALSE) | It searches for pattern in x. | st1 <<br>c('abcd','bdcd','abcdabcd')<br>pattern<- '^abc'<br>print(grep(pattern, st1))<br>**Output**<br>[1] 1 3 |
| **3.** | sub(pattern, replacement, x ignore.case =FALSE fixed=FALSE) | It finds pattern in x and replaces it with replacemen (new) text. | st1<- "England is beautifu but no the part of EU"<br>sub("England', "UK", st1)<br>**Output**<br>[1] "UK is beautiful but no a part of EU" |
| **4.** | paste(..., sep="") | It concatenates strings afte using sep string to separate them. | paste('one',2,'three',4,'five')<br>**Output**<br>[1] one 2 three 4 five |
| **5.** | strsplit(x, split) | It splits the elements o character vector x at spli point. | a<-"Split all the character"<br>print(strsplit(a, ""))<br>**Output**<br>[[1]]<br>[1] "split" "all" "the'<br>"character" |
| **6.** | tolower(x) | It is used to convert the string into lower case. | st1<- "shuBHAm"<br>print(tolower(st1))<br>**Output**<br>[1] shubham |
| **7.** | toupper(x) | It is used to convert the string into upper case. | st1<- "shuBHAm"<br>print(toupper(st1))<br>**Output**<br>[1] SHUBHAM |

### Statistical Probability Functions

R provides various statistical probability functions to perform statistical task. These statistical functions are very helpful to find normal density, normal quantile and many more calculation. In R, there are following functions which are used:

| S. No | Function | Description | Example |
|---|---|---|---|
| **1.** | dnorm(x, m=0, sd=1, log=False) | It is used to find the height o the probability distribution a | a <- seq(-7, 7, by=0.1) |

| | | | |
|---|---|---|---|
| | | each point to a given mean and standard deviation | b <- dnorm(a, mean=2.5 sd=0.5)<br>png(file="dnorm.png")<br>plot(x,y)<br>dev.off() |
| **2.** | pnorm(q, m=0, sd=1 lower.tail=TRUE, log.p=FALSE) | it is used to find the probability of a normally distributed random numbers which are less than the value of a given number. | a <- seq(-7, 7, by=0.2)<br>b <- dnorm(a, mean=2.5 sd=2)<br>png(file="pnorm.png")<br>plot(x,y)<br>dev.off() |
| **3.** | qnorm(p, m=0, sd=1) | It is used to find a number whose cumulative value matches with the probability value. | a <- seq(1, 2, by=002)<br>b <- qnorm(a, mean=2.5 sd=0.5)<br>png(file="qnorm.png")<br>plot(x,y)<br>dev.off() |
| **4.** | rnorm(n, m=0, sd=1) | It is used to generate random numbers whose distribution is normal. | y <- rnorm(40)<br>png(file="rnorm.png")<br>hist(y, main="Normal Distribution")<br>dev.off() |

### Other Statistical Function

Apart from the functions mentioned above, there are some other useful functions which helps for statistical purpose. There are the following functions:

| S. No | Function | Description | Example |
|---|---|---|---|
| **1.** | mean(x, trim=0 na.rm=FALSE) | It is used to find the mean for x object | a<-c(0:10, 40)<br>xm<-mean(a)<br>print(xm)<br>**Output**<br>[1] 7.916667 |
| **2.** | sd(x) | It returns standard deviation of an object. | a<-c(0:10, 40)<br>xm<-sd(a)<br>print(xm)<br>**Output**<br>[1] 10.58694 |
| **3.** | median(x) | It returns median. | a<-c(0:10, 40)<br>xm<-meadian(a)<br>print(xm)<br>**Output** |

| | | | [1]  5.5 |
|---|---|---|---|
| **4.** | quantilie(x, probs) | It returns quantile where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0, 1] | |
| **5.** | range(x) | It returns range. | a<-c(0:10, 40)<br>xm<-range(a)<br>print(xm)<br>**Output**<br>[1]  0  40 |
| **6.** | sum(x) | It returns sum. | a<-c(0:10, 40)<br>xm<-sum(a)<br>print(xm)<br>**Output**<br>[1]  95 |

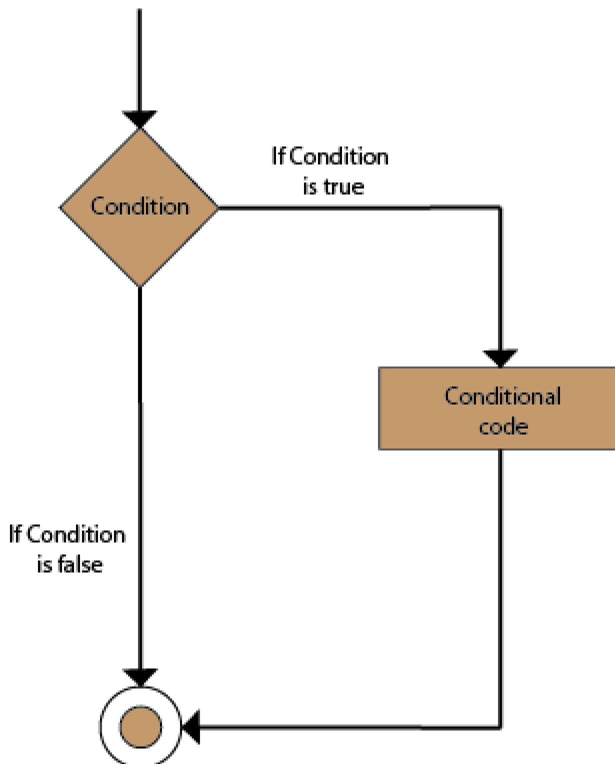
## R if Statement

The if statement consists of the Boolean expressions followed by one or more statements. The if statement is the simplest decision-making statement which helps us to take a decision on the basis of the condition.

The if statement is a conditional programming statement which performs the function and displays the information if it is proved true.

The block of code inside the if statement will be executed only when the boolean expression evaluates to be true. If the statement evaluates false, then the code which is mentioned after the condition will run.

**Flow Chart**




The syntax of if statement in R is as follows:

```
if(boolean_expression) {
  // If the boolean expression is true, then statement(s) will be executed.
}
```

Example 1:

```
x <-24L
y <- "shubham"
if(is.integer(x))
{
   print("x is an Integer")
}
```

**Output:**

```
Command Prompt

C:\Users\ajeet\R>Rscript Keyword.R
24 is an Integer
```

Example 2:

```
x <-20
y<-24
count=0
if(x<y)
{
   cat(x,"is a smaller number\n")
   count=1
}
if(count==1){
   cat("Block is successfully execute")
}
```

**Output:**

```
Command Prompt

C:\Users\ajeet\R>Rscript If.R
20 is a smaller number
Block is successfully execute
C:\Users\ajeet\R>
```

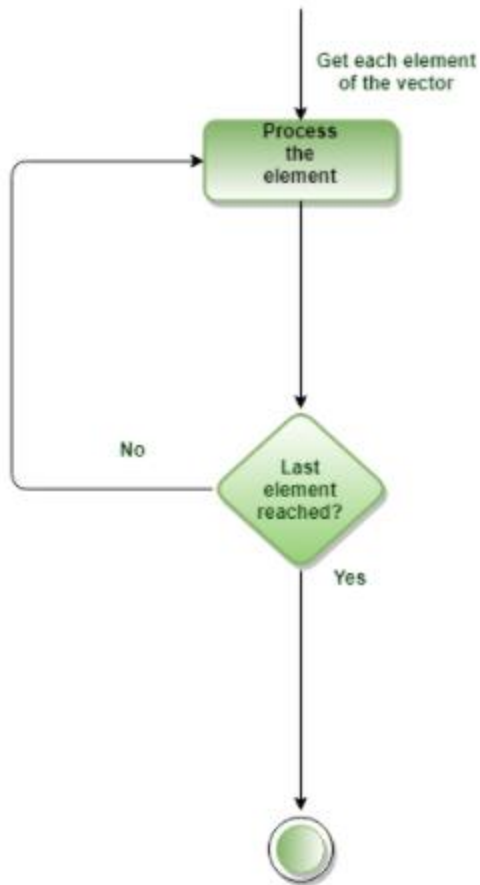
## R For Loop

A for loop is the most popular control flow statement. A for loop is used to iterate a vector. It is similar to the while loop. There is only one difference between for and while, i.e., in while loop, the condition is checked before the execution of the body, but in for loop condition is checked after the execution of the body.

In R, a for loop is a way to repeat a sequence of instructions under certain conditions. It allows us to automate parts of our code which need repetition. In simple words, a for loop is a repetition control structure. It allows us to efficiently write the loop that needs to execute a certain number of time.

Syntax:

```
for (value in vector) {
   statements
}
```




Example1:

```
# Create fruit vector
fruit <- c('Apple', 'Orange',"Guava", 'Pinapple', 'Banana','Grapes')
# Create the for statement
for ( i in fruit){
   print(i)
}
```

**Output**

```
Select Command Prompt
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet>cd R

C:\Users\ajeet\R>Rscript for.R
[1] "Apple"
[1] "Orange"
[1] "Guava"
[1] "Pinapple"
[1] "Banana"
[1] "Grapes"
```

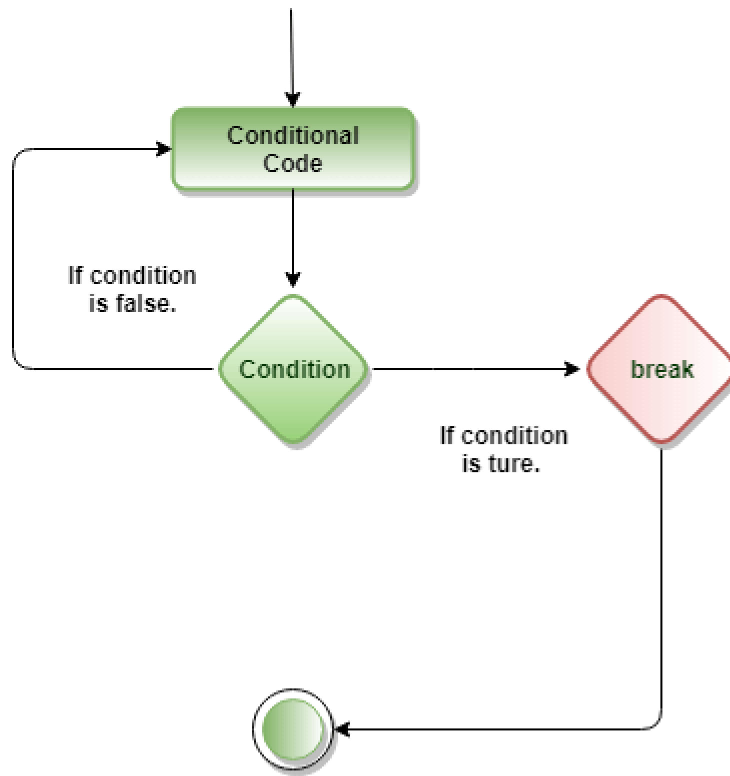
### R repeat loop

A repeat loop is used to iterate a block of code. It is a special type of loop in which there is no condition to exit from the loop. For exiting, we include a break statement with a user-defined condition. This property of the loop makes it different from the other loops.

A repeat loop constructs with the help of the repeat keyword in R. It is very easy to construct an infinite loop in R.

The basic syntax of the repeat loop is as follows:

```
repeat {
  commands
  if(condition) {
    break
  }
}
```




1. First, we have to initialize our variables than it will enter into the Repeat loop.
2. This loop will execute the group of statements inside the loop.
3. After that, we have to use any expression inside the loop to exit.
4. It will check for the condition. It will execute a break statement to exit from the loop
5. If the condition is true.
6. The statements inside the repeat loop will be executed again if the condition is false.

Example 1:

```
v <- c("Hello","repeat","loop")
cnt <- 2
repeat {
  print(v)
  cnt <- cnt+1

  if(cnt > 5) {
    break
  }
}
```

Output:

```
[1] "Hello"  "repeat" "loop"
[1] "Hello"  "repeat" "loop"
[1] "Hello"  "repeat" "loop"
[1] "Hello"  "repeat" "loop"
```

Example 2:

```
a <- 1
repeat {
  if(a == 10)
    break
  if(a == 7){
    aa=a+1
    next
  }
  print(a)
  a <- a+1
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 8
[1] 9
```

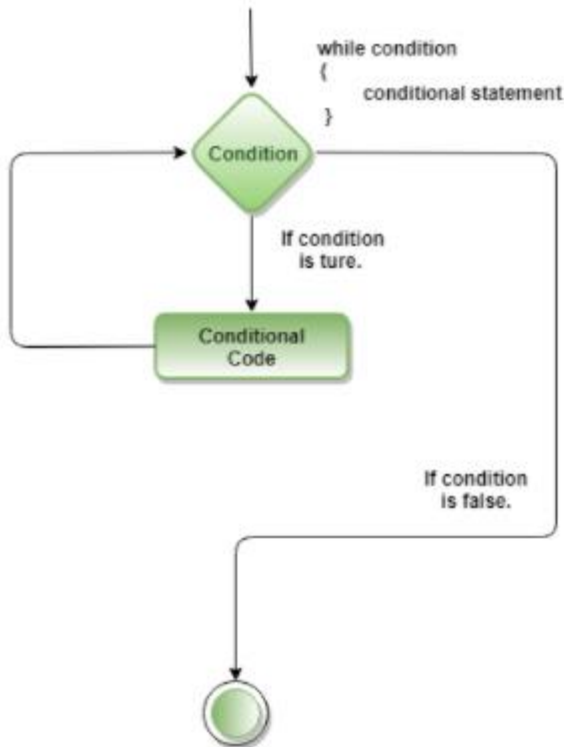
### R while loop

A while loop is a type of control flow statements which is used to iterate a block of code several numbers of times. The while loop terminates when the value of the Boolean expression will be false.

In while loop, firstly the condition will be checked and then after the body of the statement will execute. In this statement, the condition will be checked n+1 time, rather than n times.

The basic syntax of while loop is as follows:

```
while (test_expression) {
  statement
}
```




Example 1:

```
v <- c("Hello","while loop","example")
cnt <- 2
while (cnt < 7) {
  print(v)
  cntcnt = cnt + 1
}}
```

Output:

```
[1] "Hello"      "while loop" "example"
[1] "Hello"      "while loop" "example"
[1] "Hello"      "while loop" "example"
[1] "Hello"      "while loop" "example"
[1] "Hello"      "while loop" "example"
```

## A short list of some useful R commands

help() #give help regarding a command, e.g. help(hist)
c() #concatenate objects, e.g. x = c(3,5,8,9) or y = c(”Jack”,”Queen”,”King”)
1:19 #create a sequence of integers from 1 to 19
(…) #give arguments to a function, e.g. sum(x), or help(hist)
[…] #select elements from a vector or list, e.g. x[2] gives 5, x[c(2,4)] gives 5 9 for x as above
matrix() #fill in (by row) the values from y in a matrix of 4 rows and 3 columns by giving
#m = matrix(y,4,3,byrow=T)
dim() #gives the number of rows and the number of columns of a matrix, or a data frame
head() #gives the first 6 rows of a large matrix, or data frame
tail() #gives the last 6 rows of a large matrix, or data frame

```
m[ ,3] #gives the 3rd column of the matrix m
m[2, ] #gives the 2nd row of the matrix m
= or <- #assign something to a variable, e.g. x = c(”a”,”b”,”b”,”e”)
== #ask whether two things are equal, e.g. x = c(3,5,6,3) and then x == 3 gives T F F T
#Then y[x == 3] gives those entries of y where x equals 3, i.e. the 1st and 4th entry of y
< #ask whether x is smaller than y, e.g. x < 6 in the example above gives True True False True
> #ask whether x is larger than y
& #logical „and“
| #logical „or“
sum() #get the sum of the values in x by sum(x)
mean() #get the mean of the values in x by mean(x)
median() #get the median of the values in x by median(x)
sd() #get the standard deviation of the values in x
var() #get the variance of the values in x
IQR() #get the IQR of the values in x
summary() #get the summary statistics of a single variable, or of all variables in a data frame
round() #round values in x to 3 decimal places by round(x,3)
sort() #sort the values in x by giving sort(x)
unique() #get the non-duplicate values from a list, e.g. x = c(3,5,7,2,3,5,9,3) and then
unique(x) #gives 3 5 7 2 9
length(x) #gives the length of the vector x, which is 8
hist() #create a histogram of the values in x by hist(x)
stem() #create a stem and leaf plot of the values in x by stem(x)
boxplot() #create a boxplot of the values in x by boxplot(x)
plot() #scatterplot of x vs. y by plot(x,y); for more parameters see help(plot.default)
cor() #gives the linear correlation coefficient
lm() #fit a least squares regression of y (response) on x (predictor) by fit = lm(y~x)
names() #get or set the names of elements in a R object. E.g. names(fit) will give the names of the
R
#object named “fit”, or
#get or set the names of variables in a data frame.
fit$coef #gives the least squares coefficients from the fit above, i.e. intercept and slope
fit$fitted #gives the fitted values for the regression fitted above
fit$residuals #gives the residuals for the regression fitted above
lines() #add a (regression) line to a plot by lines(x,fit$fitted)
abline() #add a straight line to a scatterplot
points() #add additional points (different plotting character) to a plot by points(x,y2,pch=5)
scan() #read data for one variable from a text file, e.g. y = scan(”ping.dat”)
#Don“t forget to change to the appropriate directory first
read.table() #read spreadsheet data (i.e. more than one variable) from a text file
table() #frequency counts of entries, ideally the entries are factors(although
 #it works with integers or even reals)
write() #write the values of a variable y in a file data.txt by write(y,file=”data.txt”)
log() #natural logarithm (i.e. base e)
log10() #logarithm to base 10
seq() #create a sequence of integers from 2 to 11 by increment 3 with seq(2,11,by=3)
rep() #repeat n times the value x, e.g. rep(2,5) gives 2 2 2 2 2
getwd() #get the current working directory.
```

```
setwd() #change the directory to. E.g. setwd("c:/RESEARCH/GENE.project/Chunks/")
dir() #list files in the current working directory
search() #searching through reachable datasets and packages
library() #link to a downloaded R package to the current R session. E.g. library(Biostrings) link to the
#R package #called "Biostrings" which you had downloaded earlier onto your laptop
```

### Input and Display

```
load("c:/RData/pennstate1.RData") #load a R data frame
read.csv(filename="c:/stat251/ui.csv",header=T) #read .csv file with labels in first row
x=c(1,2,4,8,16) #create a data vector with specified elements
y=c(1:10) #create a data vector with elements 1-10
vect=c(x,y) #combine them into one vector of length 2n
mat=cbind(x,y) #combine them into a n x 2 matrix
mat[4,2] #display the 4th row and the 2nd column
mat[3,] #display the 3rd row
mat[,2] #display the 2nd column
subset(dataset,logical) #those objects meeting a logical criterion
subset(data.df,select=variables,logical) #get those objects from a data frame that meet a
#logical criterion
data.df[data.df=logical] #yet another way to get a subset
x[order(x$B),] #sort a dataframe by the order of the elements in B
x[rev(order(x$B)),] #sort the dataframe in reverse order
```

### Moving Around

```
ls() #list the R objects in the current workspace
rm(x) #remove x from the workspace
rm(list=ls()) #remove all the variables from the workspace
attach(mat) #make the names of the variables in the matrix or data frame
 #available in the workspace
detach(mat) #releases the names
new=old[,-n] #drop the nth column
new=old[-n,] #drop the nth row
new=subset(old,logical) #select those cases that meet the logical condition
complete = subset(data.df,complete.cases(data.df)) #find those cases with no missing values
new=old[n1:n2,n3:n4] #select the n1 through n2 rows of variables n3 through n4)
```

### Data Manipulation

```
x.df=data.frame(x1,x2,x3 ...) #combine different kinds of data into a data frame
scale() #converts a data frame to standardized scores
round(x,n) #rounds the values of x to n decimal places
ceiling(x) #vector x of smallest integers > x
floor(x) #vector x of largest interger < x
as.integer(x) #truncates real x to integers (compare to round(x,0)
as.integer(x < cutpoint) #vector x of 0 if less than cutpoint, 1 if greater than cutpoint)
factor(ifelse(a < cutpoint, "Neg", "Pos")) #is another way to dichotomize and to make a factor for analysis
transform(data.df,variable names = some operation) #can be part of a set up for a data set
```

### Statistical Tests

```
binom.test()
```

prop.test() #perform test with proportion(s)
t.test() #perform t test
chisq.test() #perform Chi-square test
pairwise.t.test()
power.anova.test()
power.t.test()
aov()
anova()
TukeyHSD()
kruskal.test()

### Distributions

sample(x, size, replace = FALSE, prob = NULL) # take a simple random sample of size n from the
# population x with or without replacement
rbinom(n,size,p)
pbinom()
qbinom()
dbinom()
rnorm(n,mean,sd) #randomly generate n numbers from a Normal distribution with the specific mean and sd
pnorm() #find probability (area under curve) of a Normal(10,3^2) distribution to the left
#of 8,i.e. P(X <= 8), by pnorm(8,mean=10,sd=3)
qnorm() #find quantity or value x such that area under Normal(10,3^2) curve and to the left
#of x equals 0.25 by qnorm(0.25,mean=10,sd=3)
rt()
pt()
qt()
runif(n,lower,upper)
punif()
qunif()

## Graphics with the lattice package

The lattice package provides a comprehensive graphical system for visualizing univariate and multivariate data. In particular, many users turn to the lattice package because of its ability to easily generate trellis graphs. A trellis graph displays the distribution of a variable or the relationship between variables, separately for each level of one or more other variables.
Lattice is an add-on package that implements Trellis graphics (originally developed for S and S-PLUS) in R.It is a powerful and elegant high-level data visualization system, with an emphasis on multivariate data, that is sufficient for typical graphics needs, and is also flexible enough to handle most nonstandard requirements.
The lattice package is a graphics and data visualization package inspired by the trellis graphics package. The main focus of the package is multivariate data. It has a wide variety of functions that enable it to create basic plots of the base R package as well as enhance on them. Let us start looking at all the functions and graphs in the lattice package, one-by-one.
The package provides better defaults. It also provides the ability to display multivariate relationships and it improves on the base-R graphics. This package supports the creation of trellis graphs:

- graphs that display a variable or

- the relationship between variables, conditioned on one or
- other variables.

The typical format is:

**graph_type(formula, data=)**

| graph_type | description | formula examples | Description |
|---|---|---|---|
| barchart | bar chart | x~A or A~x | |
| bwplot | boxplot | x~A or A~x | |
| cloud | 3D scatterplot | z~x*y\|A | |
| contourplot | 3D contour plot | z~x*y | |
| densityplot | kernal density plot | ~x\|A*B | y~x \| A*B means display the relationship between numeric variables y and x separately for every combination of factor A and B levels. |
| dotplot | dotplot | ~x\|A | ~x\|A means display numeric variable x for each level of factor A |
| histogram | histogram | ~x | ~x means display numeric variable x alone. |
| levelplot | 3D level plot | z~y*x | |
| parallel | parallel coordinates plot | data frame | |
| splom | scatterplot matrix | data frame | |
| stripplot | strip plots | A~x or x~A | |
| xyplot | scatterplot | y~x\|A | |
| wireframe | 3D wireframe graph | z~y*x | |

| Options | Description |
|---|---|
| aspect | A number specifying the aspect ratio (height/width) for the graph in each panel. |
| col, pch, lty, lwd | Vectors specifying the colors, symbols, line types, and line widths to be used in plotting, respectively. |
| group | Grouping variable (factor) |
| index.cond | List specifying the display order of the panels |
| key (or auto.key) | Function used to supply legend(s) for grouping variable(s). |
| layout | Two-element numeric vector specifying the arrangement of the panels (number of columns, number of rows). If desired, a third element can be added to indicate the number of pages. |
| main, sub | Character vectors specifying the main title and subtitle |

| panel | Function used to generate the graph in each panel. |
|---|---|
| scales | List providing axis annotation information |
| strip | Function used to customize panel strips. |
| split, position | Numeric vectors used to place more than one graph on a page. |
| type | Character vector specifying one or more plotting options for scatter plots (p=points, l=lines, r=regression line, smooth=loess fit, g=grid, and so on). |
| xlab, ylab | Character vectors specifying horizontal and vertical axis labels. |
| xlim, ylim | Two-element numeric vectors giving the minimum and maximum values for the horizontal and vertical axes, respectively. |

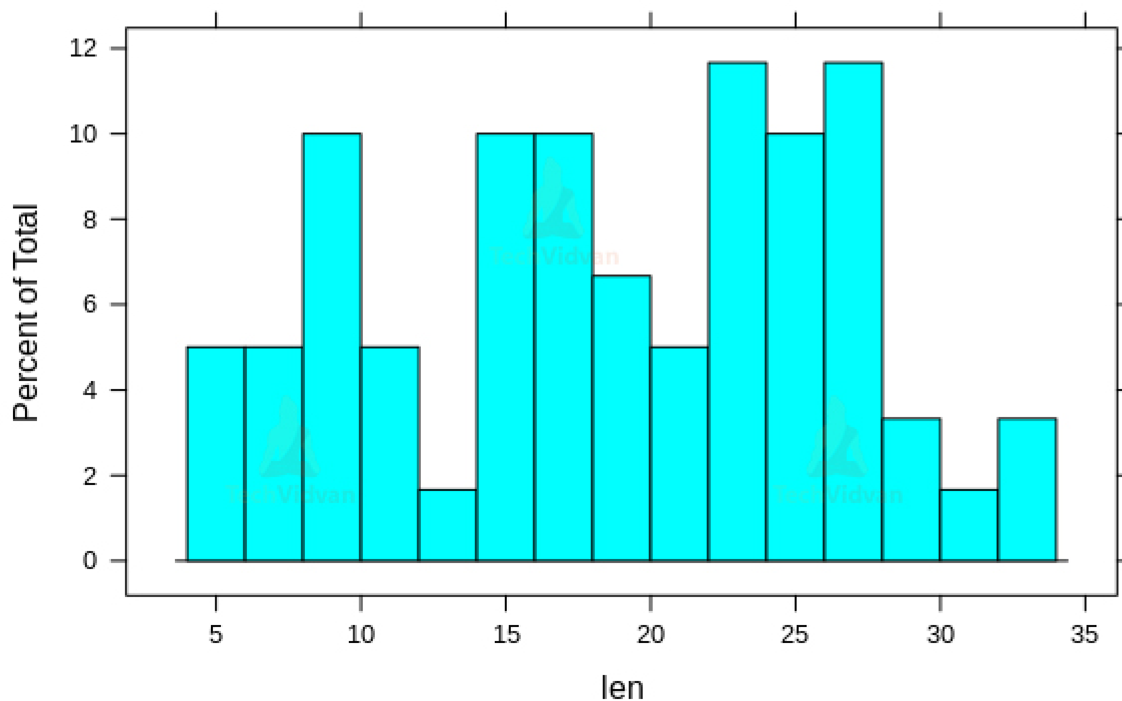
1. **Histograms in the Lattice Package**

   To create a histogram using the lattice package, we can use the histogram() function.

   Example 1:

```
histogram(~ len, data = ToothGrowth,
          breaks = 20)
```

   Output:



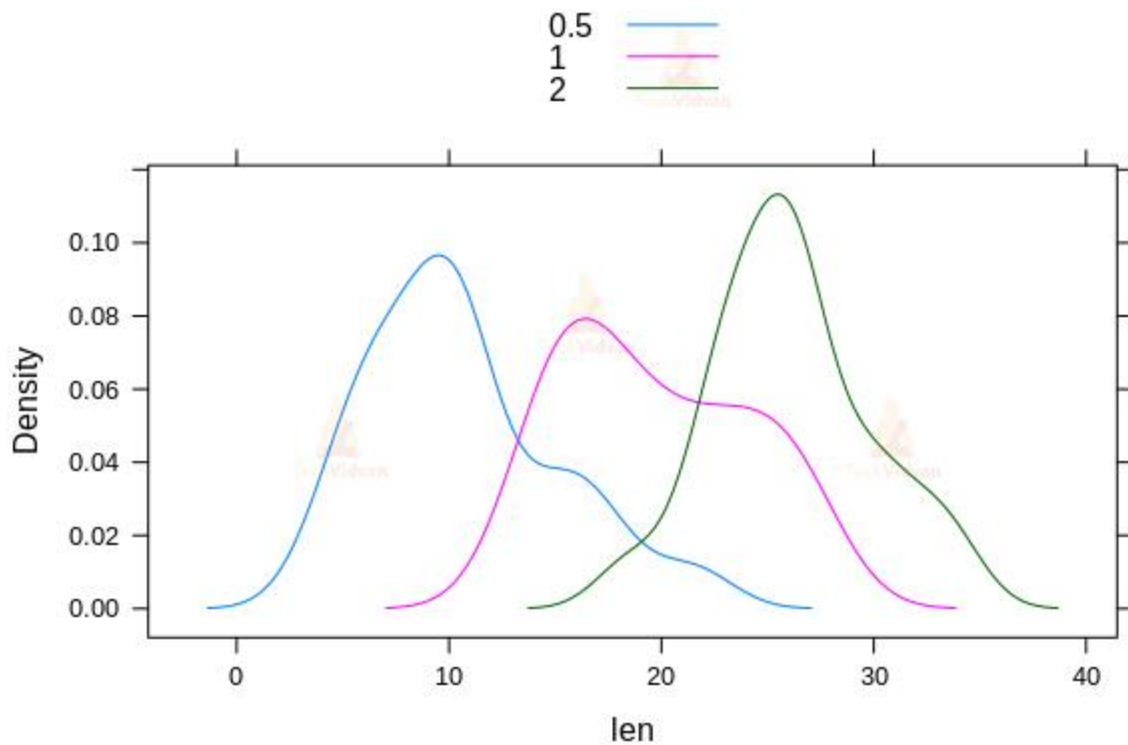

2. **Density Plots in the Lattice Package**

   We can create density plots in R using the densityplot() function of the lattice package

   Example 1:

```
densityplot(~ len, data = ToothGrowth,
            plot.points = FALSE,
            groups = dose,
            auto.key = TRUE)
```

   Output:



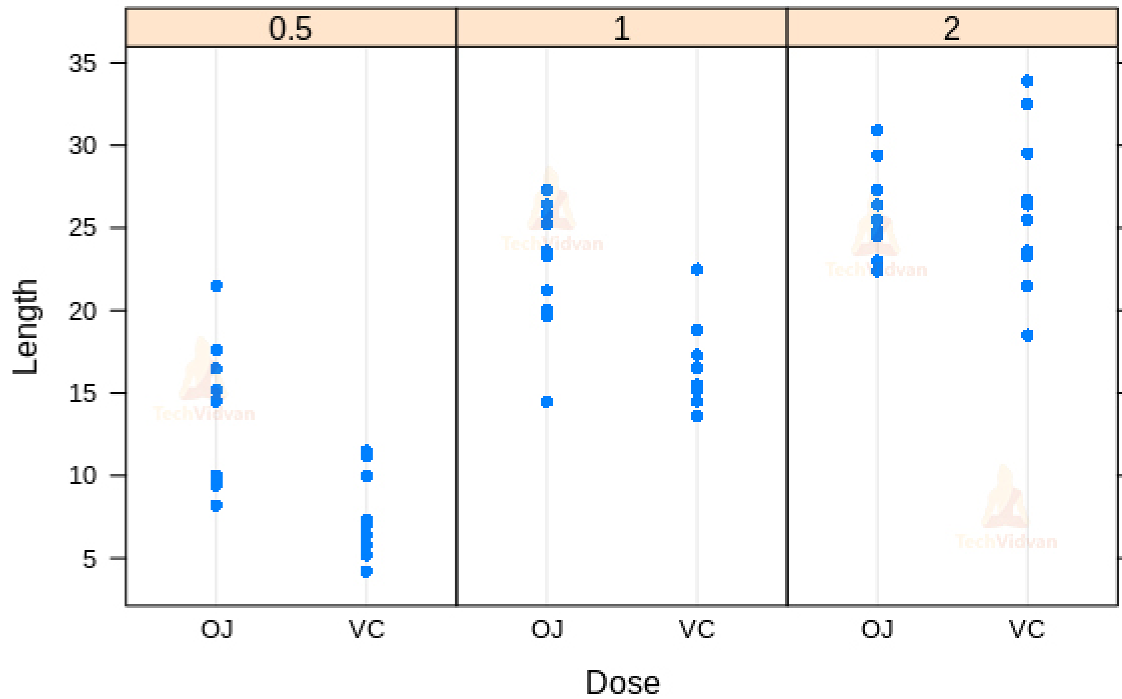

3. **Dotplots in Lattice Package in R**

The R dotplot() function enables us to create dot plots in R.

Example 1:

```
dotplot(len ~ supp|dose,
        data = ToothGrowth,
        layout = c(3,1),
        xlab = "Dose",
        ylab = "Length")
```

Output:



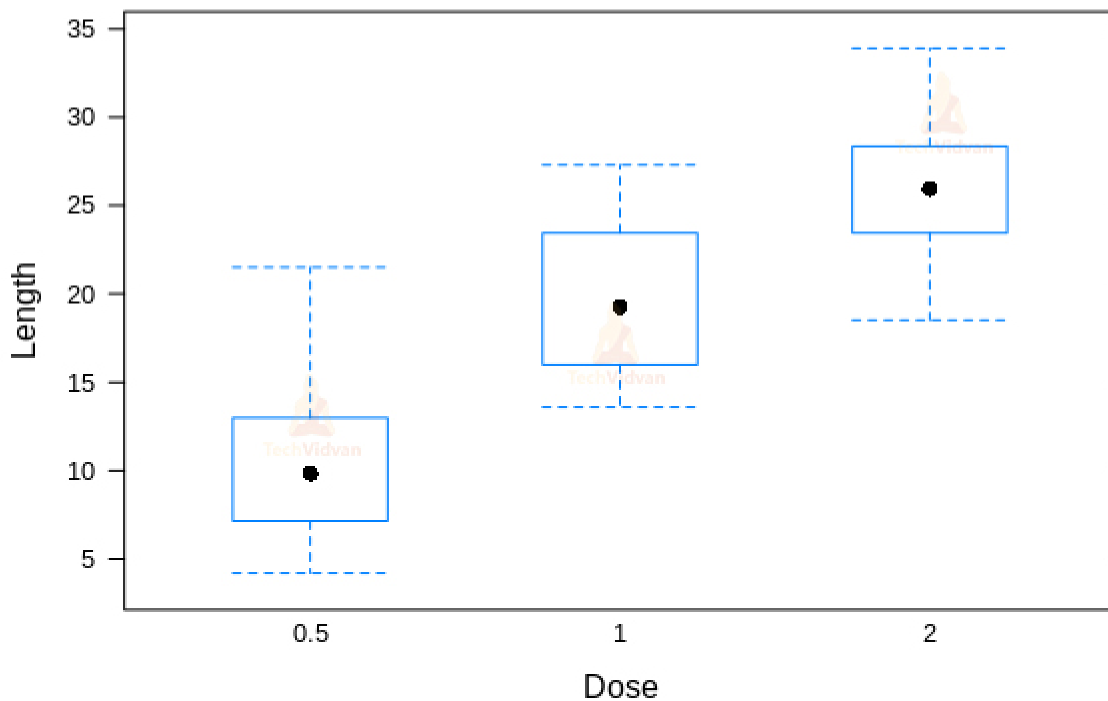

4. **Boxplots in Lattice Package in R**

The lattice package provides a bwplot() function that can be used to create a box plot.

Example:

```
ToothGrowth$dose <- as.factor(ToothGrowth$dose)
bwplot(len ~ dose, data = ToothGrowth,
       xlab = "Dose",
       ylab = "Length")
```

Output:

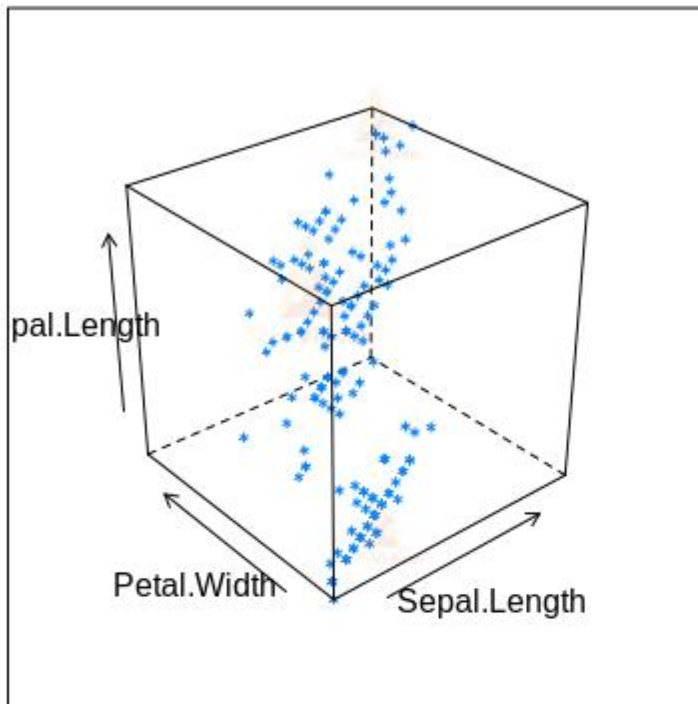



5. **3D scatter plots**

Using the cloud() function, we can create a 3D scatter plot.
Example:

```
cloud(Sepal.Length ~ Sepal.Length*Petal.Width,
      data = iris)
```

Output:

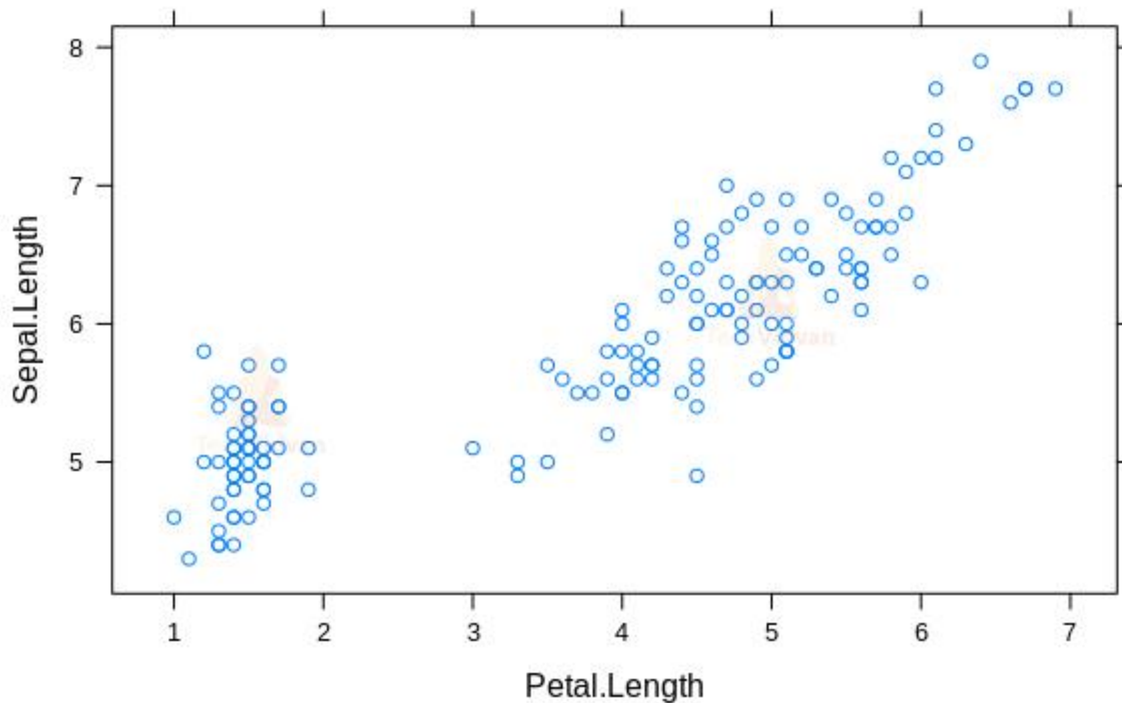



6. **Scatter Plots in the Lattice Package**

The xyplot() function can be used to create a scatter plot in R using the lattice package.
Example:

```
library(lattice)
xyplot(Sepal.Length ~ Petal.Length,
       data = iris)
```

Output:




## Unit-3 Standard Statistical Models in R & Statistics with R

standard statistical models in R and understand how they play a crucial role in data analysis. Here are the key points:

### Introduction to R and Basics in Statistics:

R is an open-source programming language designed for statistical computing and data analysis.
It provides a rich ecosystem of packages for various tasks.
You can find introductory lecture notes on using R for graphics and basic statistical analyses here1.

### Defining Statistical Models in R:

R offers an interlocking suite of facilities for fitting statistical models.
The basic output is minimal, and you can extract details using specific functions.
Formulas play a crucial role in defining statistical models.

### Packages for Standard Statistical Functionality:

R packages cover linear models, classical tests, high-level plotting functions, and survival analysis.
Explore packages like lm, glm, and survival for modeling and analysis.
You can install additional packages directly from the R prompt.

### Learning Statistics with R:

Consider reading "Learning Statistics with R," which covers introductory statistics using R2.
Learn data manipulation, scripting, and practical applications.
standard statistical models in R and delve into the fascinating world of data analysis. Here are the key points, paragraph by paragraph:

### Introduction to R and Basics in Statistics:

R is an open-source programming language and environment designed for statistical computing and data analysis.
Developed from the S language, R provides a rich ecosystem of packages for various tasks.

## Defining Statistical Models in R:

R simplifies fitting statistical models through an interlocking suite of facilities.The basic output is minimal; details are obtained by calling extractor functions.
The template for a statistical model involves linear regression with independent, homoscedastic errors.
Formulas play a crucial role in defining models.

## Examples of Model Specifications:

y ~ x and y ~ 1 + x both imply simple linear regression of y on x.
y ~ 0 + x and y ~ -1 + x perform regression through the origin (without an intercept).
log(y) ~ x1 + x2 represents multiple regression of the transformed variable log(y) on x1 and x2.
y ~ poly(x, 2) and y ~ 1 + x + I(x^2) involve polynomial regression of y on x (degree 2).
Explore interactions, classifications, and nested models using various formulae.

output and extraction from fitted models in the context of linear regression. I'll break it down for you:

Simple Linear Regression:
Purpose: Simple linear regression estimates the relationship between two quantitative variables using a straight line.
Example: Suppose you're researching the relationship between income and happiness. You survey 500 people, asking them to rank their happiness (dependent variable) on a scale from 1 to 10 based on their income (independent variable).
Model: The model is a straight line: happiness = $\beta_0 + \beta_1$(income).
Assumptions:
Homogeneity of variance (homoscedasticity): Error size doesn't change significantly across income levels.
Independence of observations: Data collected using valid sampling methods.
Normality: Data follows a normal distribution.
Linearity: The relationship between income and happiness is linear.
Fitted Model: Extracted coefficients: happiness = -11.89 + 0.98(income).
Extracting Fitted Values:
After fitting the model, we want to predict the response variable (happiness) for each observation.
Use the fitted.values attribute from the model object.
Example: Suppose you have data on minutes played and fouls committed by basketball players. You fit a multiple linear regression model to predict points scored:
Model: points = $\beta_0 + \beta_1$(minutes) + $\beta_2$(fouls)
Extract fitted values: Add a new column in your data frame with the fitted values.
Here's how you can extract fitted values in R:

R

```
# Create data frame (df) with minutes, fouls, and points
df <- data.frame(minutes = c(5, 10, 13, 14, 20, 22, 26, 34,
38, 40),
                 fouls = c(5, 5, 3, 4, 2, 1, 3, 2, 1, 1),
                 points = c(6, 8, 8, 7, 14, 10, 22, 24, 28,
30))

# Fit multiple linear regression model
fit <- lm(points ~ minutes + fouls, data = df)

# Extract fitted values into a new column
df$fitted <- fit$fitted.values
```

LinearRegression

Example 1: Simple Linear Regression

Dataset Description:

We have data on income (scaled to match happiness scores) and happiness ratings (on a scale of 1 to 10) from an imaginary sample of 500 people.

Steps:

Load the Data into R:

Download the simple regression dataset.

In RStudio, go to File > Import dataset > From Text (base).

Choose the downloaded file (income.data) and import it.

Perform Simple Linear Regression:

```
# Load necessary packages
install.packages("ggplot2")
install.packages("dplyr")
install.packages("broom")
install.packages("ggpubr")
library(ggplot2)
library(dplyr)
library(broom)

# Load the dataset
income_data <- read.table("income.data", header = TRUE)

# Fit the model
linearMod <- lm(happiness ~ income, data = income_data)

# Summary of the model
summary(linearMod)
```

he output will show coefficients, standard errors, t-values, and p-values for the intercept and income predictor.

Interpret the coefficients to understand the relationship between income and happiness.

Check Assumptions and Visualize Results:

Ensure homoscedasticity (constant variance of residuals).

Create scatter plots and regression lines using ggplot2.

Example 2: Multiple Linear Regression

**Dataset Description:**

We have data on the percentage of people biking to work, smoking, and heart disease prevalence in an imaginary sample of 500 towns.

Steps:

Load the Data into R:

Download the multiple regression dataset.

Import the data into RStudio as done in Example

```
# Load necessary packages (if not already loaded)
library(ggplot2)
library(dplyr)
library(broom)

# Load the dataset
heart_data <- read.table("heart.data", header = TRUE)

# Fit the model
multiMod <- lm(heart.disease ~ biking + smoking, data = heart_data)

# Summary of the model
summary(multiMod)
```

Interpret the coefficients for biking and smoking predictors.

Assess model fit using R-squared and F-statistic.

logistic regression in R

Logistic regression is a powerful technique for binary classification problems, where the response variable (Y) can take only two values (e.g., 0 or 1, Yes or No). I'll provide step-by-step instructions along with code snippets.

**Example 1: Predicting Default Status**

Dataset Description:

We'll use the Default dataset from the ISLR package. This dataset contains information about 10,000 individuals, including whether they defaulted on their credit card payments, student status, average balance, and income.

Steps:

Load the Data:

Load the Default dataset and view a summary:

```
# Load dataset
data <- ISLR::Default
# View summary
summary(data)
```

Create Training and Test Sets:

Split the dataset into a training set (70%) and a testing set (30%)

```
# Make this example reproducible
set.seed(1)
# Split data
sample <- sample(c(TRUE, FALSE), nrow(data), replace = TRUE, prob =
c(0.7, 0.3))
train <- data[sample, ]
test <- data[!sample, ]
```

Fit the Logistic Regression Model:

Use the glm function with family = "binomial"

```
# Fit logistic regression model
model <- glm(default ~ student + balance + income, family = "binomial",
data = train)
# Disable scientific notation for model summary
options(scipen = 999)
# View model summary
summary(model)
```

**Example 2: Heart Disease Prediction**

Dataset Description:

We'll use a heart disease dataset.

Features include sex, years addicted, and comorbid drug use.

Steps:

Load the Data:

Import and clean the heart disease dataset (convert factors, handle missing values, etc.).

Perform Logistic Regression:

Start with one independent variable (e.g., sex) and interpret the coefficients, log odds, log odds ratio, deviance residuals, and p-values.

Then perform logistic regression with multiple independent variables and interpret summary output (residual deviance, AIC, pseudo R-squared, overall p-value).

Graph Predicted Probabilities:

Compare predicted probabilities of heart disease with actual patient status.

## Linear Mixed-Effect Models (LMMs) in R with two examples.

LMMs are powerful for dissecting hierarchical and longitudinal data by accounting for random effects and addressing correlated residuals and heterogeneous variance. Here are the examples:

**Example: Educational Level and Income Relationship**

Problem: Suppose you want to study the relationship between average income ((y)) and educational levels in a town comprising four fully segregated blocks.

Issue: If you model this using classic linear models, you neglect dependencies among observations within the same block, leading to correlated residuals.

Solution: Linear mixed-effect models (LMMs) can handle this spatial correlation by introducing random effects that account for differences among random samples.

Reference: 1

**Example: Anxiety Levels and Blood Measurements**

Problem: Suppose you want to study the relationship between anxiety levels ((y)) and triglyceride and uric acid levels in blood samples from 1,000 people measured 10 times over 24 hours.

Issue: Classic linear models may not address the changing variance of (y) over time (heteroscedasticity).

Solution: LMMs can handle heterogeneous variance using specific variance functions, improving estimation accuracy and interpretation of fixed effects.

**Summarizing Data IN R**

Summarizing data in R is a common task during data analysis. Let's explore how to use the dplyr package to group and summarize data. I'll provide examples using the built-in dataset called mtcars.

1. Install & Load the dplyr Package: Before using dplyr, make sure you have it installed

```
# Install dplyr (if not already installed)
install.packages('dplyr')
# Load dplyr
library(dplyr)
```

**Group and Summarize Data**: The basic syntax for grouping and summarizing data is as follows

```
data %>%
  group_by(col_name) %>%
  summarize(summary_name = summary_function)
```

**Example 1: Calculate Mean & Median by Group (Cylinder):**

```
# Find mean mpg by cylinder
mtcars %>%
  group_by(cyl) %>%
  summarize(mean_mpg = mean(mpg, na.rm = TRUE))

# Find median mpg by cylinder
mtcars %>%
  group_by(cyl) %>%
  summarize(median_mpg = median(mpg, na.rm = TRUE))
```

**Example 2: Calculate Measures of Spread by Group (Cylinder):**

```
# Find standard deviation (sd), interquartile range (IQR), and median
absolute deviation (mad) by cylinder
mtcars %>%
  group_by(cyl) %>%
  summarize(sd_mpg = sd(mpg, na.rm = TRUE),
            iqr_mpg = IQR(mpg, na.rm = TRUE),
            mad_mpg = mad(mpg, na.rm = TRUE))
```

**Example 3: Count Rows by Group (Cylinder)**

```
# Find row count and unique row count by cylinder
mtcars %>%
  group_by(cyl) %>%
  summarize(count_mpg = n(),
            u_count_mpg = n_distinct(mpg))
```

**Example 4: Find Percentile by Group (Cylinder)**

```r
# Find the 90th percentile of mpg by cylinder group
mtcars %>%
  group_by(cyl) %>%
  summarize(percentile_90_mpg = quantile(mpg, probs = 0.9, na.rm = TRUE))
```

**-Calculating Relative Frequencies**

Let's calculate relative frequencies using the dplyr package in R. I'll provide examples based on the given data frame.

**Relative Frequency of One Variable (Team)**:

```r
# Load the dplyr package (if not already loaded)
library(dplyr)

# Create a sample data frame
df <- data.frame(
  team = c('A', 'A', 'A', 'B', 'B', 'B', 'B'),
  position = c('G', 'F', 'F', 'G', 'G', 'G', 'F'),
  points = c(12, 15, 19, 22, 32, 34, 39)
)

# Calculate relative frequencies by team
df %>%
  group_by(team) %>%
  summarise(n = n()) %>%
  mutate(freq = n / sum(n))
```

This tells us that team A accounts for 42.9% of all rows in the data frame, while team B accounts for the remaining 57.1%.

**Relative Frequency of Multiple Variables (Team and Position)**

```r
# Calculate relative frequencies by team and position
df %>%
  group_by(team, position) %>%
  summarise(n = n()) %>%
  mutate(freq = n / sum(n))
```

66.7% of players on team A are in position F.
33.3% of players on team A are in position G.

25.0% of players on team B are in position F.
75.0% of players on team B are in position G.

**Display Relative Frequencies as Percentages**:

```
# Calculate relative frequencies and display as percentages
df %>%
  group_by(team, position) %>%
  summarise(n = n()) %>%
  mutate(freq = paste0(round(100 * n / sum(n), 0), '%'))
```

Team A: 67% in position F, 33% in position G.
Team B: 25% in position F, 75% in position G.

**TabulatingFactors and Creating Contingency Tables**

Creating contingency tables (also known as cross-tabulation or crosstabs) in R allows us to summarize and explore the relationship between two categorical variables. These tables provide insights into how categories within one variable intersect with those in another. Let’s dive into creating and interpreting contingency tables:

**What is a Contingency Table?**
A contingency table summarizes the joint distribution of two categorical variables. Unlike frequency tables that provide a one-dimensional view of categorical data, contingency tables offer a multidimensional perspective.

Example Data
Suppose we have a dataset representing 20 different product orders. Each order includes the type of product purchased (e.g., TV, Radio, Computer) and the country where the product was purchased (A, B, C, D).

```
# Create example data
df <- data.frame(
  order_num = 1:20,
  product = rep(c('TV', 'Radio', 'Computer'), times = c(9, 6, 5)),
  country = rep(c('A', 'B', 'C', 'D'), times = 5)
)

# View the data
df
```

**Creating a Contingency Table in R**

We can use the table() function to create a contingency table by providing the variables product and country as arguments:

```
# Create contingency table
contingency_table <- table(df$product, df$country)

# View the contingency table
contingency_table
```

The resulting table shows the number of specific products ordered from each country:

```
         A B C D
Computer 1 1 1 2
Radio    1 2 2 1
TV       3 2 2 2
```

**Interpreting the Contingency Table**

The value in the bottom right corner (9 + 6 + 5 = 20) represents the total number of products ordered.

The row sums (right side) show the total number of each product type ordered: 5 computers, 6 radios, and 9 TVs.

The column sums (bottom) show the total number of products ordered from each country: 5 from A, 5 from B, 5 from C, and 5 from D.

Contingency tables help us understand relationships between categorical variables and identify patterns.

## Testing Categorical Variables for Independence

To test the independence of categorical variables in R, you can use the Chi-Square Test of Independence. This statistical test helps determine whether there is a significant association between two categorical variables. Let's walk through an example using R:

**Example: Chi-Square Test of Independence**

Suppose we want to investigate whether gender is associated with political party preference. We collect data from a simple random sample of 500 voters and survey them on their political party preference. The results are summarized in the following table:

**Table**

| | Republican | Democrat | Independent | Total |
|---|---|---|---|---|
| Male | 120 | 90 | 40 | 250 |
| Female | 110 | 95 | 45 | 250 |
| Total | 230 | 185 | 85 | 500 |

Here are the steps to perform the Chi-Square Test of Independence in R:

Create the Data: First, create a table to hold the data

```
# Create the data
data <- matrix(c(120, 90, 40, 110, 95, 45), ncol = 3, byrow = TRUE)
colnames(data) <- c("Rep", "Dem", "Ind")
rownames(data) <- c("Male", "Female")
data <- as.table(data)
```

Perform the Chi-Square Test: Next, use the chisq.test() function to perform the Chi-Square Test of Independence:

```
# Perform Chi-Square Test of Independence
result <- chisq.test(data)
```

Interpret the Output: The output provides the following information:

Chi-Square Test Statistic: 0.86404

Degrees of Freedom: 2 (calculated as (#rows - 1) * (#columns - 1))

p-value: 0.6492

Since the p-value (0.6492) is not less than 0.05, we fail to reject the null hypothesis. This means we do not have sufficient evidence to say that there is an association between gender and political party preference. In other words, gender and political party preference are independent.

Remember to adjust the significance level (alpha) based on your specific research context.

HOW TO PERFORM Calculating Quantiles of a Dataset

Calculating quantiles of a dataset in R is straightforward using the quantile() function. Quantiles divide a ranked dataset into equal groups, providing valuable insights into the data distribution. Let's explore how to calculate quartiles (a specific type of quantile) in R:

**Example 1: Calculate Quartiles of a Vector**

Suppose we have the following dataset representing customer spendings:

```
# Customer spendings data
customer_spendings <- c(120, 150, 200, 230, 250, 300, 350)
```

To calculate quartiles (25th, 50th, and 75th percentiles), use the quantile() function:

The output will show the quartiles:

```
# Calculate quartiles
quartiles <- quantile(customer_spendings, prob = c(0.25, 0.5, 0.75))
print(quartiles)
```

**Example 2: Calculate Quartiles of Columns in a Data Frame**

Suppose we have a data frame with two columns: var1 and var2. We want to calculate quartiles for each column:

The output will provide quartile values for each column.

**Example 3: Calculate Quantiles by Group**

Suppose we have a data frame with team names and points. We want to calculate quartiles by team:

```
# Create a data frame
df <- data.frame(
  var1 = c(9, 9, 8, 9, 10, 9, 3, 5, 6, 8, 9, 10, 11, 12, 13, 11, 10),
  var2 = c(7, 7, 8, 3, 2, 6, 8, 9, 11, 11, 16)
)

# Calculate quartiles of column 'var2'
quantile(df$var2, prob = c(0.25, 0.5, 0.75))

# Alternatively, calculate quartiles for all columns
sapply(df, function(x) quantile(x, prob = c(0.25, 0.5, 0.75)))
```

## HOW TO Converting Data in to z-scores in r

Converting data into z-scores (standardized scores) in R is a common task for comparing distributions and understanding how individual values relate to the mean and standard deviation. Let's explore how to calculate z-scores for different scenarios:

**Z-Scores for a Single Vector**: Suppose you have a vector of data points. You can calculate the z-score for each value using the following formula: [ z = \frac{{X - \mu}}{{\sigma}} ] where:

(X) is the raw data value.

(\mu) is the population mean.

(\sigma) is the population standard deviation.

Here's an example using a vector of data:

```
# Create a vector of data
data <- c(6, 7, 7, 12, 13, 13, 15, 16, 19, 22)

# Calculate z-scores
z_scores <- (data - mean(data)) / sd(data)
z_scores
```

The resulting z_scores tell you how many standard deviations each value is away from the mean.

Z-Scores for a Single Column in a DataFrame: If you have a data frame with multiple columns, you can calculate z-scores for a specific column. For example:

```r
# Create a data frame
df <- data.frame(
  Pressure = c(78.2, 88.2, 71.7, 80.21, 84.21, 82.56, 72.12, 73.85),
  Temperature = c(35, 36, 37, 38, 32, 30, 31, 34)
)

# Calculate z-scores for the 'Pressure' column
z_scores <- (df$Pressure - mean(df$Pressure)) / sd(df$Pressure)
z_scores
```

**Z-Scores for Every Column in a DataFrame**: To calculate z-scores for every column in a data frame, you can use the sapply() function:

Each column in z_scores_df contains z-scores relative to its own mean and standard deviation.

**HOW TO PERFORM T-TEST**

Performing a t-test in R is a common statistical analysis to compare means between two groups. Let's walk through how to conduct different types of t-tests using R:

**One-Sample T-Test:**

The one-sample t-test compares the mean of a sample to a known population mean (hypothesized mean).

Example: Suppose we have a sample of mouse weights, and we want to test if the average weight differs from a hypothesized value (e.g., 25 grams).

.

```r
# Example data (mice weights)
mice_weights <- c(18.9, 19.5, 23.1, 20.2, 21.8, 22.3, 19.8, 20.5, 21.7,
22.9)

# One-sample t-test
t_test_result <- t.test(mice_weights, mu = 25)

# Interpretation
cat("One-Sample T-Test Results:\n")
cat("Test Statistic (t):", t_test_result$statistic, "\n")
cat("P-value:", t_test_result$p.value, "\n")
cat("95% Confidence Interval:", t_test_result$conf.int, "\n")
```

**Two-Sample (Independent) T-Test:**

Compares the means of two independent groups (e.g., treatment vs. control).

Example: Comparing the average sales performance of two marketing teams.

```
# Example data (sales performance)
team_a_sales <- c(120, 130, 140, 125, 135)
team_b_sales <- c(100, 110, 105, 115, 120)

# Two-sample t-test
t_test_result <- t.test(team_a_sales, team_b_sales)

# Interpretation
cat("Two-Sample T-Test Results:\n")
cat("Test Statistic (t):", t_test_result$statistic, "\n")
cat("P-value:", t_test_result$p.value, "\n")
cat("95% Confidence Interval:", t_test_result$conf.int, "\n")
```

**Paired (Dependent) T-Test:**

Compares means of paired observations (e.g., before vs. after treatment).

Example: Analyzing blood pressure measurements before and after a drug treatment.

```
# Example data (blood pressure)
bp_before <- c(120, 125, 130, 122, 128)
bp_after <- c(115, 118, 123, 118, 125)

# Paired t-test
t_test_result <- t.test(bp_before, bp_after, paired = TRUE)

# Interpretation
cat("Paired T-Test Results:\n")
cat("Test Statistic (t):", t_test_result$statistic, "\n")
cat("P-value:", t_test_result$p.value, "\n")
cat("95% Confidence Interval:", t_test_result$conf.int, "\n")
```

## Testing  Sample Proportions

how to test sample proportions in R using the prop.test() function. This function allows you to perform hypothesis tests and construct confidence intervals for proportions. I'll provide examples for both one-sample and two-sample proportion tests:

**One-Sample Proportion Test:**

The one-sample proportion test compares a sample proportion to a known population proportion or a hypothesized proportion.

Example: Suppose we have 107 trials with 42 successes. We want to test if the proportion of successes is equal to 0.6 at a 95% confidence level.

```
# Hypothesis test for a single proportion
prop.test(x = 42, n = 107, p = 0.6, conf.level = 0.95)
```

**Two-Sample Proportion Test**:

The two-sample proportion test compares proportions between two independent groups.

Example: Comparing the proportions of success in two marketing teams (Team A and Team B).

```
# Example data (success counts)
team_a_success <- 120
team_a_total <- 250
team_b_success <- 110
team_b_total <- 250

# Two-sample proportion test
prop.test(c(team_a_success, team_b_success), c(team_a_total, team_b_total))
```

## Testing  Normality IN R

Testing for normality is essential before applying many statistical methods. In R, there are several ways to check if your data follows a normal distribution. Let's explore some common methods:

**Histogram:**

Create a histogram of your data. If it resembles a bell-shaped curve, it suggests normality.

Example

```r
# Generate example data (replace with your own)
normal_data <- rnorm(200)
non_normal_data <- rexp(200, rate = 3)

# Create histograms
par(mfrow = c(1, 2))
hist(normal_data, col = 'steelblue', main = 'Normal')
hist(non_normal_data, col = 'steelblue', main = 'Non-normal')
```

**Q-Q Plot (Quantile-Quantile Plot):**

Plot the quantiles of your data against the quantiles of a theoretical normal distribution. If the points fall along a straight diagonal line, your data is likely normal.

Example

```r
par(mfrow = c(1, 2))
qqnorm(normal_data, main = 'Normal')
qqline(normal_data)
qqnorm(non_normal_data, main = 'Non-normal')
qqline(non_normal_data)
```

**Shapiro-Wilk Test:**

A formal statistical test that assesses normality. If the p-value is greater than 0.05, the data is assumed to be normally distributed.

Example:

```r
shapiro.test(normal_data)
shapiro.test(non_normal_data)
```

**Comparing Means of Two Samples**

Comparing means of two samples in R involves various statistical tests, including t-tests. Let's explore how to perform different types of mean comparisons using R:

**One-Sample T-Test:**

The one-sample t-test compares the mean of a sample to a known population mean or a hypothesized mean.

Example: Suppose we have a sample of mouse weights, and we want to test if the average weight differs from a hypothesized value (e.g., 25 grams).

```
# Example data (mouse weights)
mouse_weights <- c(18.9, 19.5, 23.1, 20.2, 21.8, 22.3, 19.8, 20.5, 21.7,
22.9)

# One-sample t-test
t_test_result <- t.test(mouse_weights, mu = 25)

# Interpretation
cat("One-Sample T-Test Results:\n")
cat("Test Statistic (t):", t_test_result$statistic, "\n")
cat("P-value:", t_test_result$p.value, "\n")
cat("95% Confidence Interval:", t_test_result$conf.int, "\n")
```

**Two-Sample (Independent) T-Test:**

Compares the means of two independent groups (e.g., treatment vs. control).

Example: Comparing the average sales performance of two marketing teams.

```
# Example data (sales performance)
team_a_sales <- c(120, 130, 140, 125, 135)
team_b_sales <- c(100, 110, 105, 115, 120)

# Two-sample t-test
t_test_result <- t.test(team_a_sales, team_b_sales)

# Interpretation
cat("Two-Sample T-Test Results:\n")
cat("Test Statistic (t):", t_test_result$statistic, "\n")
cat("P-value:", t_test_result$p.value, "\n")
cat("95% Confidence Interval:", t_test_result$conf.int, "\n")
```

**Paired (Dependent) T-Test:**

Compares means of paired observations (e.g., before vs. after treatment).

Example: Analyzing blood pressure measurements before and after a drug treatment.

```r
# Example data (blood pressure)
bp_before <- c(120, 125, 130, 122, 128)
bp_after <- c(115, 118, 123, 118, 125)

# Paired t-test
t_test_result <- t.test(bp_before, bp_after, paired = TRUE)

# Interpretation
cat("Paired T-Test Results:\n")
cat("Test Statistic (t):", t_test_result$statistic, "\n")
cat("P-value:", t_test_result$p.value, "\n")
cat("95% Confidence Interval:", t_test_result$conf.int, "\n")
```

### Testing Correlation for Significance

Testing the significance of a correlation coefficient in R involves statistical tests to determine if the observed correlation is statistically different from zero. Let's explore how to perform this using an example dataset:

Example Data: Suppose we have two numeric vectors representing the number of mobile phones sold by two sales teams (Team A and Team B) in a week:

```r
# Example data
team_a_sales <- c(120, 130, 140, 125, 135)
team_b_sales <- c(100, 110, 105, 115, 120)
```

**Scatterplot Visualization**: Before performing the test, let's create a scatterplot to visualize the relationship between the two variables:

```r
# Create a scatterplot
plot(team_a_sales, team_b_sales, pch = 16, col = 'blue', main = 'Sales Comparison')
```

The scatterplot will help us understand the direction of the relationship.

**Correlation Test:** We'll perform a correlation test using the cor.test() function

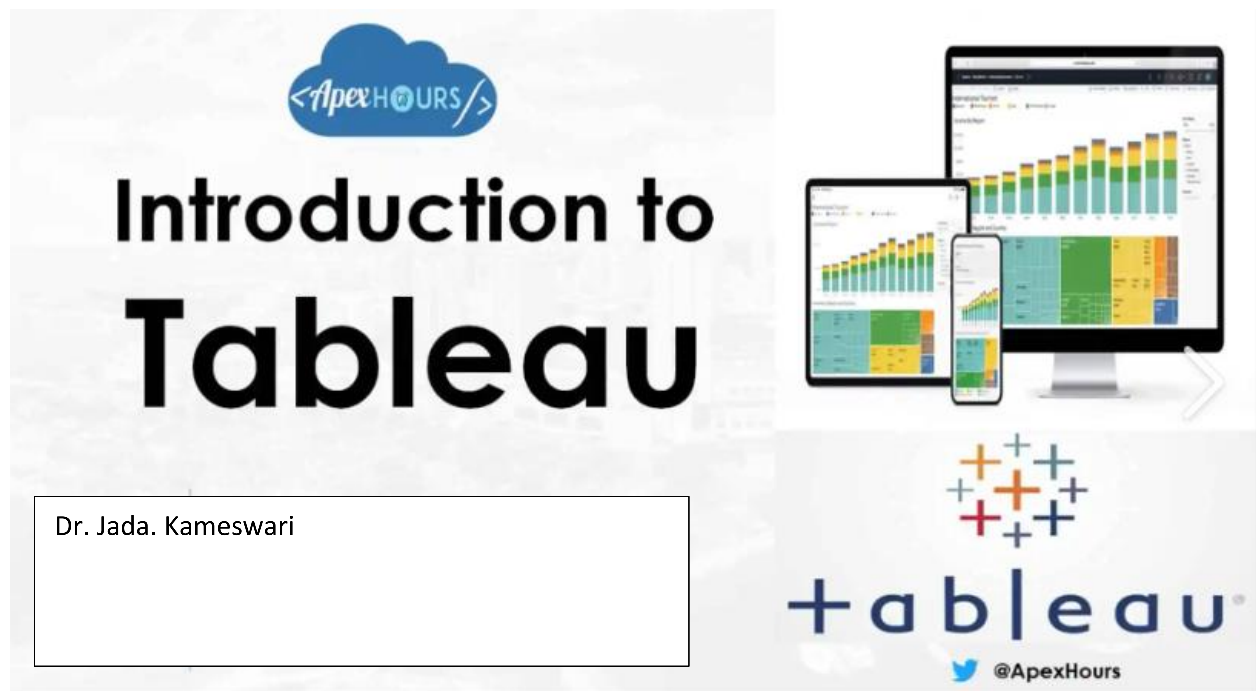
```
# Perform correlation test
correlation_result <- cor.test(team_a_sales, team_b_sales)

# Interpretation
cat("Correlation Test Results:\n")
cat("Test Statistic (t):", correlation_result$statistic, "\n")
cat("P-value:", correlation_result$p.value, "\n")
cat("95% Confidence Interval:", correlation_result$conf.int, "\n")
```

The resulting p-value will indicate whether the correlation is statistically significant. If the p-value is less than the chosen significance level (e.g., 0.05), we reject the null hypothesis that the correlation is zero.

Conclusion: Based on the p-value, we can determine whether there is a significant correlation between the sales performance of Team A and Team B.

## Chapter IV
TABLEAU




Introduction to Tableau

Tableau is a powerful data visualization tool used for converting raw data into an understandable format. It helps in simplifying raw data into an easily understandable format without any technical skills and coding knowledge.

Key Features of Tableau

Data Blending: Combine data from multiple sources.

Real-time Analysis: Analyze data in real-time.

Collaboration of Data: Share data visualizations with others.

Getting Started with Tableau

Connecting to Data: Tableau can connect to various data sources like Excel, SQL Server, Google Sheets, etc.

Creating Visualizations: Drag and drop fields to create visualizations like bar charts, line charts, maps, and more.

Building Dashboards: Combine multiple visualizations into a single dashboard for a comprehensive view.

Example Visualizations

1. Bar Chart !Bar Chart

2. Line Chart !Line Chart

3. Map Visualization !Map Visualization

Steps to Create a Simple Visualization

Connect to Data: Open Tableau and connect to your data source.

Drag and Drop: Drag fields into rows and columns to create your visualization.

Customize: Use filters, colors, and labels to customize your visualization.

Save and Share: Save your workbook and share it with others.

## Terminology

Key Terminology in Tableau

Understanding the terminology in Tableau is crucial for effectively using the tool. Here are some of the most important terms:

**Workbook**: A collection of one or more worksheets and dashboards.

**Worksheet**: A single view of data. Each worksheet can be connected to a single data source.

**Dashboard:** A collection of views shown in a single location where you can compare and monitor a variety of data simultaneously.

**Story**: A sequence of visualizations that work together to convey information.

**Data Source:** The underlying data that Tableau is connected to.

**Filter:** A control on a view that limits the data shown in a view.

**Marks**: Visual representations of one or more rows in a data source. Mark types can be bar, line, square, etc.

**Pills**: Pieces of data that you want to show up on your visualization. They can be dragged to different shelves.

**Shelves**: Areas on the visualization screen where you can drop pieces of data (pills) so that Tableau can act on them. Common shelves include Columns, Rows, Pages, and Filters.

**Pane**: The row and columns areas in a view.

**Repository**: A folder located in your My Documents folder that stores workbooks.

**View**: The visual representation of your data in a worksheet or dashboard.

**Packaged Workbook**: A type of workbook that contains both the workbook and copies of the referenced local file data sources and background images.

Example Visualizations

1. Bar Chart !Bar Chart

2. Line Chart !Line Chart

3. Map Visualization !Map Visualization

Tableau User Interface: Detailed Notes

The Tableau user interface is designed to be intuitive and user-friendly, making it easy for users to create and interact with data visualizations. Here’s a detailed breakdown of the main components:

1. **Workspace Layout**

The Tableau workspace consists of several key areas:

Data Pane: Located on the left side, it displays the data sources and fields available for analysis.

Shelves: Areas where you can drag fields to create visualizations. Common shelves include Columns, Rows, Pages, Filters, and Marks.

Cards: Used to control various aspects of the visualization, such as color, size, and labels.

View: The central area where the visualization is displayed.

2. **Toolbar**

The toolbar provides quick access to common functions:

Undo/Redo: Reverse or repeat the last action.

Save: Save the current workbook.

New Worksheet/Dashboard/Story: Create new sheets, dashboards, or stories.

Show Me: Suggests different types of visualizations based on the selected data.

3. **Data Pane**

The Data Pane is divided into two sections:

Dimensions: Categorical data fields.

Measures: Numerical data fields.

4. **Shelves**

Shelves are used to build visualizations:

Columns and Rows Shelves: Define the structure of the visualization.

Filters Shelf: Apply filters to limit the data shown.

Pages Shelf: Create visualizations that show changes over time.

Marks Shelf: Customize the appearance of the data points.

5. **Cards**

Cards control various aspects of the visualization:

Color Card: Assign colors to data points.

Size Card: Adjust the size of data points.

Label Card: Add labels to data points.

Tooltip Card: Customize the information shown when hovering over data points.

6. **View**

The View is the canvas where the visualization is displayed. You can interact with the visualization by clicking, dragging, and hovering over data points.

7. **Show Me Panel**

The Show Me panel suggests different types of visualizations based on the selected data. It helps users quickly create charts, maps, and other visual representations.

**Basic Tableau Design Flow**

Creating effective visualizations and dashboards in Tableau involves a systematic design flow. Here's a detailed guide to help you understand the basic steps:

1. **Connect to Data Source**

The first step is to connect Tableau to your data source. Tableau supports a wide range of data sources, including:

**Text Files (e.g., CSV, Excel)**

Relational Databases (e.g., SQL Server, MySQL)

Cloud Databases (e.g., Google BigQuery, Amazon Redshift)

Example: !Connect to Data Source

2. **Build Data Views**

After connecting to the data source, you can start building data views (also known as reports). This involves:

Dragging and dropping fields from the Data Pane to the Rows and Columns shelves.

Creating basic visualizations like bar charts, line charts, and scatter plots.

Example: !Build Data Views

3. **Enhance the Views**

Enhance your visualizations by:

Applying filters to focus on specific data.

Using aggregations to summarize data.

Formatting the visualization with colors, labels, and borders.

Example: !Enhance the Views

4. **Create Worksheets**

Worksheets are individual sheets where you build different views of your data. Each worksheet can contain a different type of visualization.

Example: !Create Worksheets

5. **Create and Organize Dashboards**

Dashboards combine multiple worksheets into a single view. This allows you to compare and analyze different aspects of your data simultaneously. Organize your worksheets logically to make the dashboard informative and visually appealing.

Example: !Create Dashboards

6. **Create a Story**

A story in Tableau is a sequence of worksheets or dashboards that work together to convey information. Stories are useful for presenting a narrative or demonstrating how different visualizations are connected.

Basic Visualization Design with Show Me in Tableau

The Show Me panel in Tableau is a powerful feature that helps users quickly create various types of visualizations. Here's a detailed guide on how to use it effectively:

1. **Understanding the Show Me Panel**

The Show Me panel suggests different types of visualizations based on the data you have selected. It is located on the right side of the Tableau interface and can be toggled on or off using the Show Me button.

Example: !Show Me Panel

2. **Selecting Data for Visualization**

To use the Show Me panel, you first need to select the data fields you want to visualize. You can do this by dragging fields from the Data Pane to the Columns and Rows shelves or by simply clicking on the fields.

Example: !Selecting Data

3. **Choosing a Visualization Type**

Once you have selected your data, the Show Me panel will highlight the visualization types that are suitable for your data. Click on any of the highlighted options to create the visualization.

Example: !Choosing Visualization

4. **Customizing the Visualization**

After creating the visualization, you can customize it using the Marks card, Filters, and other options available in Tableau. This includes changing colors, adding labels, and adjusting the size of the marks.

Example: !Customizing Visualization

5. **Building Dashboards and Stories**

You can combine multiple visualizations into a single dashboard or create a story to present your data in a narrative format. This helps in providing a comprehensive view of your data.

### Taking Off with Tableau

When you first encounter a dataset, often the first thing you see is the raw data—numbers, dates, text, field names, and data types. Almost certainly, there are insights and stories that need to be

uncovered and told, decisions to make, and actions to take. But how do you find the significance? How do you uncover the meaning and tell the stories that are hidden in the data?

**Tableau** is an amazing platform for seeing, understanding, and making key decisions based on your data! With it, you will be able to achieve incredible data discovery, data analysis, and data storytelling. You'll accomplish these tasks and goals visually using an interface that is designed for a natural and seamless flow of thought and work.

To leverage the power of Tableau, you don't need to write complex scripts or queries. Instead, you will be interacting with your data in a visual environment where everything that you drag and drop will be translated into the necessary queries for you and then displayed visually. You'll be working in real time, so you will see results immediately, get answers as quickly as you can ask questions, and be able to iterate through potentially dozens of ways to visualize the data to find a key insight or tell a piece of the story.

This chapter introduces the foundational principles of Tableau. We'll go through a series of examples that will introduce the basics of connecting to data, exploring and analyzing the data visually, and finally putting it all together in a fully interactive dashboard. These concepts will be developed far more extensively in subsequent chapters

**The cycle of analytics**

As someone who works with and seeks to understand data, you will find yourself working within the cycle of analytics. This cycle might be illustrated as follows:

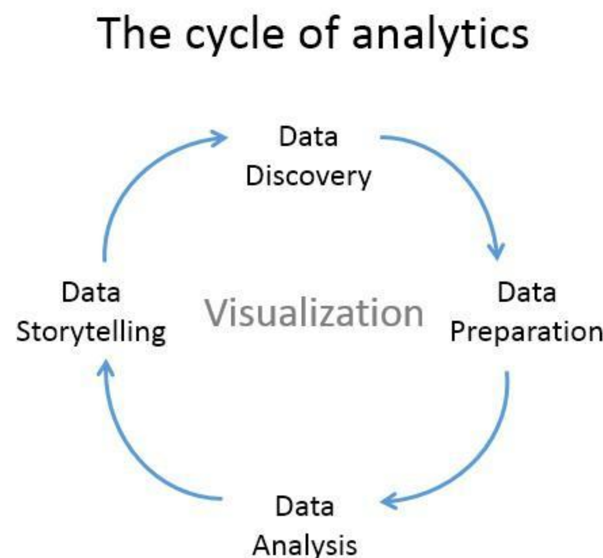


Tableau allows you to jump to any step of the cycle, move freely between steps, and iterate through the cycle very rapidly. With Tableau, you have the ability to do the following:

- **Data discovery**: You can very easily explore a dataset using Tableau and begin to understand what data you have visually.

- **Data preparation**: Tableau allows you to connect to data from many different sources and, if necessary, create a structure that works best for your analysis. Most of the time, this is as easy as pointing Tableau to a database or opening a file, but Tableau gives you the tools to bring together even complex and messy data from

multiple sources.

**Data analysis**: Tableau makes it easy to visualize the data, so you can see and understand trends, outliers, and relationships. In addition to this, Tableau has an ever-growing set of analytical functions that allow you dive deep into understanding complex relationships, patterns, and correlations in the data.

- **Data storytelling**: Tableau allows you to build fully interactive

dashboards and stories with your visualizations and insights so

that you can share the data story with others.

All of this is done visually. **Data visualization** is the heart of Tableau. You can iterate through countless ways of visualizing the data to ask and answer questions, raise new questions, and gain new insights. And you'll accomplish this as a flow of thought.

**Connecting to data**

Tableau connects to data stored in a wide variety of files and databases. This includes flat files, such as Excel documents, spatial files, and text files; relational databases, such as SQL Server and Oracle; cloud-based data sources, such as Google Analytics and Amazon Redshift; and OLAP data sources, such as Microsoft Analysis Services. With very few exceptions, the process of analysis and creating visualizations will be the same, no matter what data source you use.

We'll cover details of connecting to different types of data sources in Chapte

r 2, *Working with Data in Tableau*. And we'll cover data spanning a wide variety of industries in other chapters. For now, we'll connect to a text file, specifically, a **comma-separated values** file (.csv). The data is a variation of the sample that ships with Tableau: Superstore, a fictional retail chain that sells various products to customers across the United States. Please use the supplied data file instead of the Tableau sample data, as the variations will lead to differences in visualizations.

The Chapter 1 workbooks, included with the code files bundle, already have connections to the file, but for this example, we'll walk through the steps of creating a connection in a new workbook:

1. Open Tableau. You should see the home screen with a list of connection options on the left and, if applicable, thumbnail previews of recently edited workbooks in the center, along with sample workbooks at the bottom.
2. Under Connect and To a File, click Text File.

3. In the Open dialogue box, navigate to the \Learning Tableau\Chapter

01 directory and select the Superstore.csv file.

4. You will now see the data connection screen, which allows you to visually create connections to data sources. We'll examine the features of this screen in detail in the *Connecting to data* section of Chapter 2, *Working with Data in Tableau*. For now, Tableau has already added and given a preview of the file for the connection:

Superstore

Connection
Live Extract

Filters
0 Add

Connections Add

Superstore
Text File

Superstore.csv

Files

Superstore.csv

New Union

Sort fields Data source order

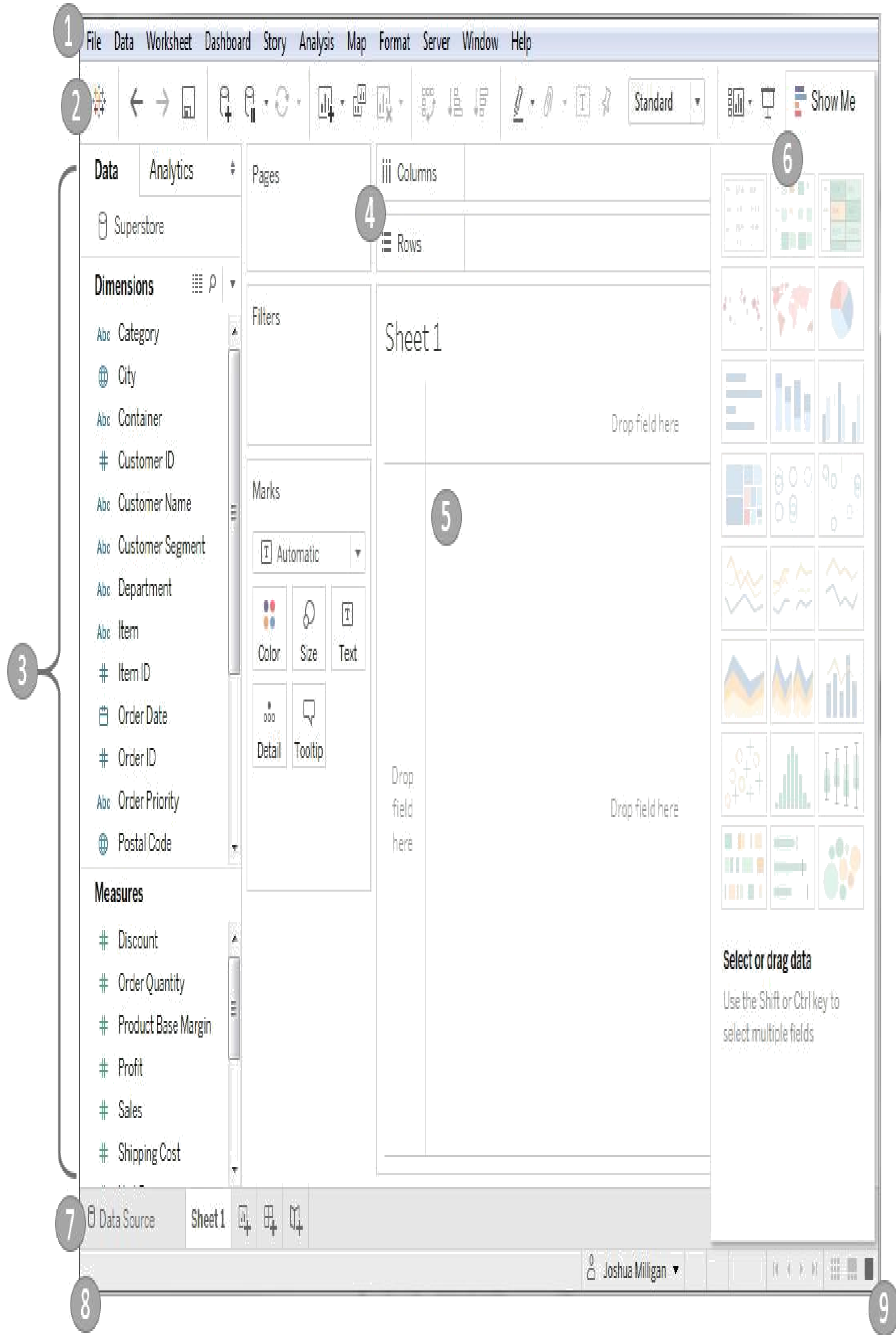
Show aliases Show hidden fields 1,000 rows

| Category | City | Container | Customer ID | Customer Name | Customer Segment |
|---|---|---|---|---|---|
| Paper | Lombard | Small Box | 3035 | Mark Bailey | Home Office |
| Paper | Lombard | Wrap Bag | 3035 | Mark Bailey | Home Office |
| Pens & Art Supplies | Southbury | Wrap Bag | 3385 | Daniel Richmond | Corporate |
| ...inder A... | Coachella | Small Box | 3133 | Kristine Singleton | Corporate |

Go to Worksheet

Data Source Sheet 1

For this connection, no other configuration is required, so simply click on the Sheet 1 tab at the bottom to start visualizing the data! You should now see the main work area within Tableau, which looks like this:


1
2
3
4
5
6
7
8
9
File
Data
Worksheet
Dashboard
Story
Analysis
Map
Format
Server
Window
Help
Standard
Show Me
Data
Analytics
Superstore
Dimensions
Category
City
Container
Customer ID
Customer Name
Customer Segment
Department
Item
Item ID
Order Date
Order ID
Order Priority
Postal Code
Measures
Discount
Order Quantity
Product Base Margin
Profit
Sales
Shipping Cost
Pages
Filters
Marks
Automatic
Color
Size
Text
Detail
Tooltip
Columns
Rows
Sheet 1
Drop field here
Drop field here
Drop field here
Select or drag data
Use the Shift or Ctrl key to select multiple fields
Data Source
Sheet 1
Joshua Milligan

We'll refer to elements of the interface throughout the book using specific terminology, so take a moment to familiarize yourself with the terms used for various components numbered in the preceding screenshot:

1. The **Menu** contains various menu items for performing a wide range of functions.
2. The **Toolbar** allows for common functions such as undo, redo, save, add a data source, and so on.
3. The **Side Bar** contains tabs for Data and Analytics. When the Data tab is active, we'll refer to the side bar as the data pane. When the Analytics tab is active, we'll refer to the side bar as the analytics pane. We'll go into detail later in this chapter, but for now, note that the data pane shows the data source at the top and contains a list of fields from the data source below, divided into Dimensions and Measures.

4. Various shelves such as Columns, Rows, Pages, and Filters serve as areas to drag and drop fields from the data pane. The **Marks** card contains additional shelves such as Color, Size, Text, Detail, and Tooltip. Tableau will visualize data based on the fields you drop on to the shelves.

*Data fields in the data pane are available to add to a view. Fields that have been dropped on to a shelf are called* **in the view** *or* **active fields** *because they play an active role in the way Tableau draws the visualization.*

5. The **canvas** or **view** is where Tableau will draw the data visualization. You may also drop fields directly on to the view. You'll find the seamless title at the top of the canvas. By default, it will display the name of the sheet, but it can be edited or even

hidden.

6. Show Me is a feature that allows you to quickly iterate through various types of visualizations based on data fields of interest. We'll look at Show Me toward the end of the chapter.
7. The tabs at the bottom of the window give you options for editing the data source, as well as navigating between and adding any number of sheets, dashboards, or stories. Many times, any tab (whether it is a sheet, a dashboard, or a story) is referred to generically as a **sheet**.

*A Tableau workbook is a collection of data sources, sheets, dashboards, and stories. All of this is saved as a single Tableau workbook file (.twb or .twbx). We'll look at the difference in file types and explore details of what else is saved as part of a workbook in later chapters. A workbook is organized into a collection of tabs of various types:*

- *A sheet is a single data visualization, such as a bar chart or a line graph. Since Sheet is also a generic term for any tab, we'll often refer to a sheet as a* **view** *because it is a single view of the data.*
- *A* **dashboard** *is a presentation of any number of related views and other elements (such as text or images) arranged together as a cohesive whole to communicate a message to an audience. Dashboards are often designed to be interactive.*
- *A* **story** *is a collection of dashboards or single views arranged to communicate a narrative from the data. Stories may also be interactive.*

8. As you work, the status bar will display important information and details about the view, selections, and the user.
9. Various controls allow you to navigate between sheets, dashboards, and stories, as well as view the tabs with **Show Filmstrip** or switch to a sheet sorter showing an interactive thumbnail of all sheets in the workbook. Now that you have connected to the data in the text file, we'll explore some examples that lay the foundation for data visualization and then move on to

building some foundational visualization types. To prepare for this, please do the following:

1. From the menu, select File | Exit.
2. When prompted to save changes, select No.
3. From the \learning Tableau\Chapter 01 directory, open the file Chapter 01 Starter.twbx. This file contains a connection to the Superstore data file and is designed to help you walk through the examples in this chapter.

*The files for each chapter include a Starter workbook that allows you to work through the examples given in this book. If at any time, you'd like to see the completed examples, open the Complete workbook for the chapter.*

With a connection to the data, you are ready to start visualizing and analyzing the data. As you begin to do so, you will take on the role of an analyst at the retail chain. You'll ask questions of the data, build visualizations to answer those questions, and ultimately design a dashboard to share the results. Let's start by laying some foundations for understanding how Tableau visualizes data.

**Foundations for building visualizations**

When you first connect to a data source such as the Superstore file, Tableau will display the data connection and the fields in the data pane on the left Side Bar. Fields can be dragged from the data pane onto the canvas area or onto various shelves such as Rows, Columns, Color, or Size. As we'll see, the placement of the fields will result in different encodings of the data based on the type of field.

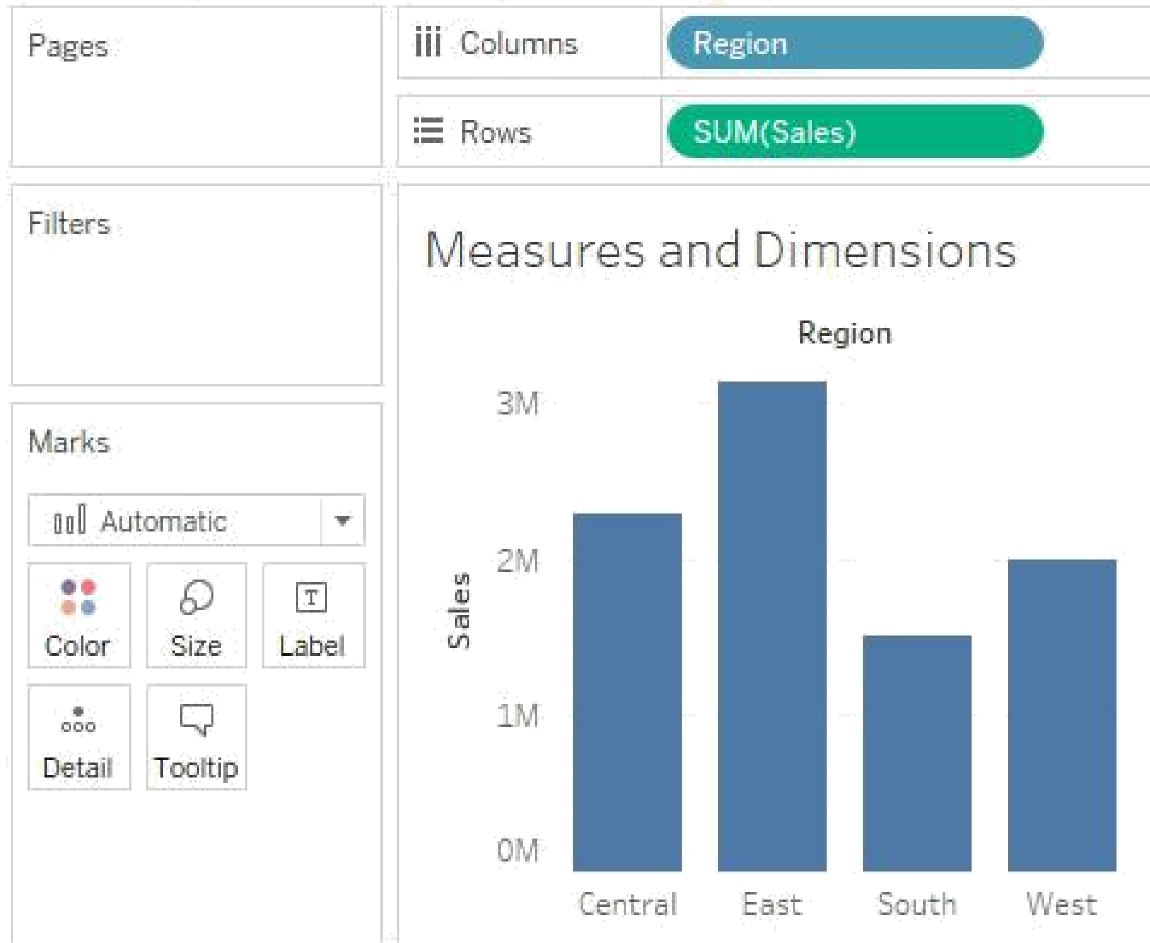
**Measures and dimensions**

The fields from the data source are visible in the data pane and are divided into **Measures** and **Dimensions**. The difference between measures and dimensions is a fundamental concept to understand when using Tableau:

Measures are values that are aggregated. For example, they are summed, averaged, counted, or have a minimum or a maximum.

Dimensions are values that determine the level of detail at which measures are aggregated. You can think of them as slicing the measures or creating groups into which the measures fit. The combination of dimensions used in the view define the view's basic level of detail.

As an example (which you can view in the Chapter 01 Starter workbook on the Measures and Dimensions sheet), consider a view created using the Region fields and Sales from the Superstore connection:




The Sales field is used as a measure in this view. Specifically, it is being aggregated as a sum. When you use a field as a measure in the view, the type aggregation (for example, SUM, MIN, MAX, and AVG) will be shown on the active field. Note that in the preceding example, the active field on rows clearly indicates the sum aggregation of Sales: SUM(Sales).

The Region field is a dimension with one of four values for each record of data: **Central**, **East**, **South**, or **West**. When the field is used as a dimension in the view, it slices the measure. So, instead of an overall sum of sales, the preceding view shows the sum of sales for each region.

**Discrete and continuous fields**

Another important distinction to make with fields is whether a field is being used as **discrete** or **continuous.** Whether a field is discrete or continuous, determines how Tableau visualizes it based on where it is used in the view. Tableau will give a visual indication of the default for a field (the color of the icon in the data pane) and how it is being used in the view (the color of the active field

on a shelf). Discrete fields, such as Region in the previous example, are blue. Continuous fields, such as Sales, are green.

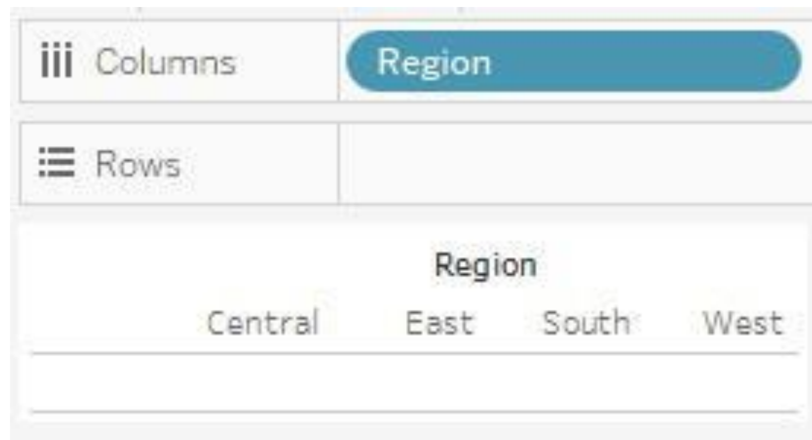
*In the screenshots in the printed version of this book, you should be able to distinguish a slight difference in shade between the discrete (blue) and the continuous (green) fields, but pay special attention to the interface as you follow along using Tableau. You may also wish to download the color image pack from Packt Publishing, available at: https://www.packtpub.com/sites/default/files/downloads/9781788839525_ColorImages.pdf*

**Discrete fields**

Discrete (blue) fields have values that are shown as distinct and separate from one another. Discrete values can be reordered and still make sense. For example, you could easily rearrange the values of Region to

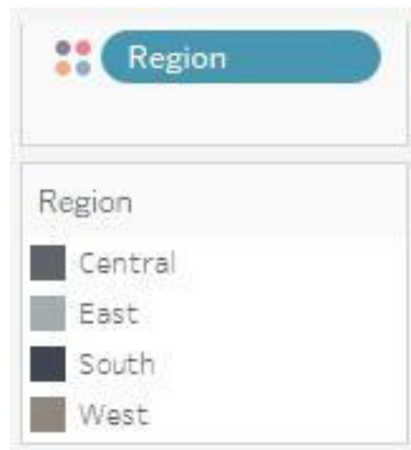
be **East**, **South**, **West**, and **Central**, instead of the default order in the preceding screenshot.

When a discrete field is used on the Rows or Columns shelves, the field defines headers. Here, the discrete field Region defines column headers:




Here, it defines row headers:

| Region |
| --- |
| Central |
| East |
| South |
| West |




When used for color, a Discrete field defines a discrete color palette in which each color aligns with a distinct value of the field:

**Continuous fields**

Continuous (green) fields have values that flow from first to last as a continuum. Numeric and date fields are often (though not always) used as continuous fields in the view. The values of these fields have an order that it would make little sense to change.

When used on Rows or Columns, a continuous field defines an axis:

When used for color, a continuous field defines a gradient:

It is very important to note that *continuous* and *discrete* are different concepts from **Measure** and **Dimension**. While most dimensions are discrete by default, and most measures are continuous by default, it is possible to use any measure as a discrete field and some dimensions as continuous fields in the view.

*To change the default of a field, right-click the field in the data pane and select Convert to Discrete or Convert to Continuous.*

*To change how a field is used in the view, right-click the field in the view and select Discrete or Continuous. Alternatively, you can drag and drop the fields between Dimensions and Measures in the data pane.*

In general, you can think of the differences between the types of fields as follows:

- Choosing between dimension and measure tells Tableau *how to slice or aggregate* the data

- Choosing between discrete and continuous tells Tableau *how to display* the data with a header or an axis and defines individual colors or a gradient.

As you work through the examples in this book, pay attention to the fields you are using to create the visualizations, whether they are dimensions or measures, and whether they are discrete or continuous. Experiment with changing fields in the view from continuous to discrete, and vice versa, to gain an understanding of the differences in the visualization.

## Visualizing data

A new connection to a data source is an invitation to explore and discover! At times, you may come to the data with very well-defined questions and a strong sense of what you expect to find. Other times, you will come to the data with general questions and very little idea of what you will find. The visual analytics capabilities of Tableau empower you to rapidly and iteratively explore the data, ask new questions, and make new discoveries.

The following visualization examples cover a few of the most foundational visualization types. As you work through the examples, keep in mind that the goal is not simply to learn how to create a specific chart. Rather, the examples are designed to help you think through the process of asking questions of the data and getting answers through iterations of visualization. Tableau is designed to make that process intuitive, rapid, and transparent.

*Something that is far more important than memorizing the steps to create a specific chart type is understanding how and why to use Tableau to create a bar chart, and adjusting your visualization to gain new insights as you ask new questions.*

### Bar charts

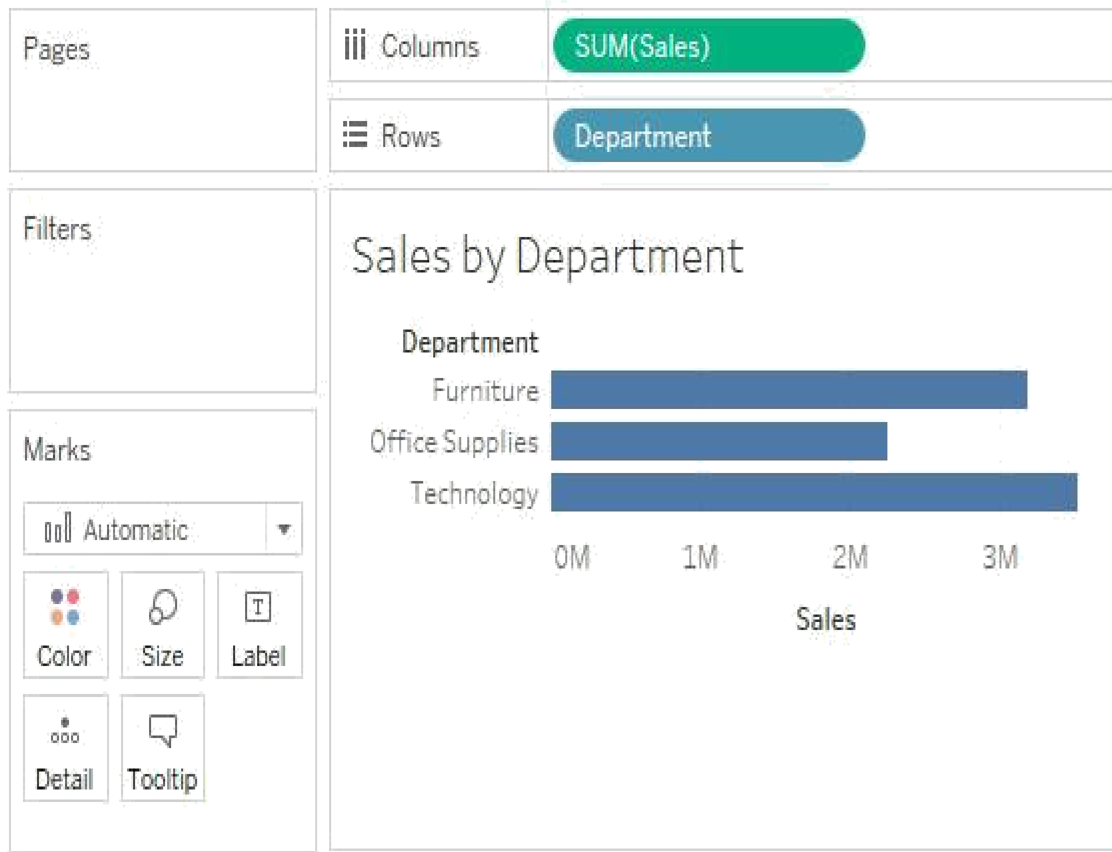
**Bar charts** visually represent data in a way that makes the comparing of values across different categories easy. The length of the bar is the primary means by which you will visually understand the data. You may also incorporate color, size, stacking, and order to communicate additional attributes and values.

Creating bar charts in Tableau is very easy. Simply drag and drop the measure you want to see onto either the Rows or Columns shelf and the dimension that defines the categories onto the opposing Rows or Columns shelf.

As an analyst for Superstore, you are ready to begin a discovery process focused on sales (especially the dollar value of sales). As you follow the examples, work your way through the sheets in the Chapter 01

Starter workbook. The Chapter 01 Complete workbook contain, the complete examples so you can compare your results at any time:

1. Click on the the Sales by Department tab to view that sheet.
2. Drag and drop the Sales field from Measures in the data pane onto the Columns shelf. You now have a bar chart with a single bar representing the sum of sales for all the data in the data source.
3. Drag and drop the Department field from Dimensions in the data pane to the Rows shelf. This slices the data to give you three bars, each having a length that corresponds to the sum of sales for each department:




You now have a horizontal bar chart. This makes comparing the sales between the departments easy. The mark type drop-down menu on the Marks card is set to Automatic and shows an indication that Tableau has determined that bars are the best visualization given the fields you have placed in the view. As a dimension, the Department slices the data. Being discrete, it defines row headers for each department in the data. As a measure, the Sales field is aggregated. Being continuous, it defines an axis. The mark type of bar causes individual bars for each department to be drawn from 0 to the value of the sum of sales for that department.

*Typically, Tableau draws a mark (such as a bar, a circle, a square) for every intersection of dimensional values in the view. In this simple case, Tableau is drawing a single bar mark for each dimensional value (***Furniture***,* **Office Supplies***, and* **Technology***) of* **Department***. The type of mark is indicated and can be changed in the drop-down menu on the Marks card. The number of marks drawn in the view can be observed on the lower-left status bar.*

Tableau draws different marks in different ways; for example, bars are drawn from 0 (or the end of the previous bar, if stacked) along the axis.

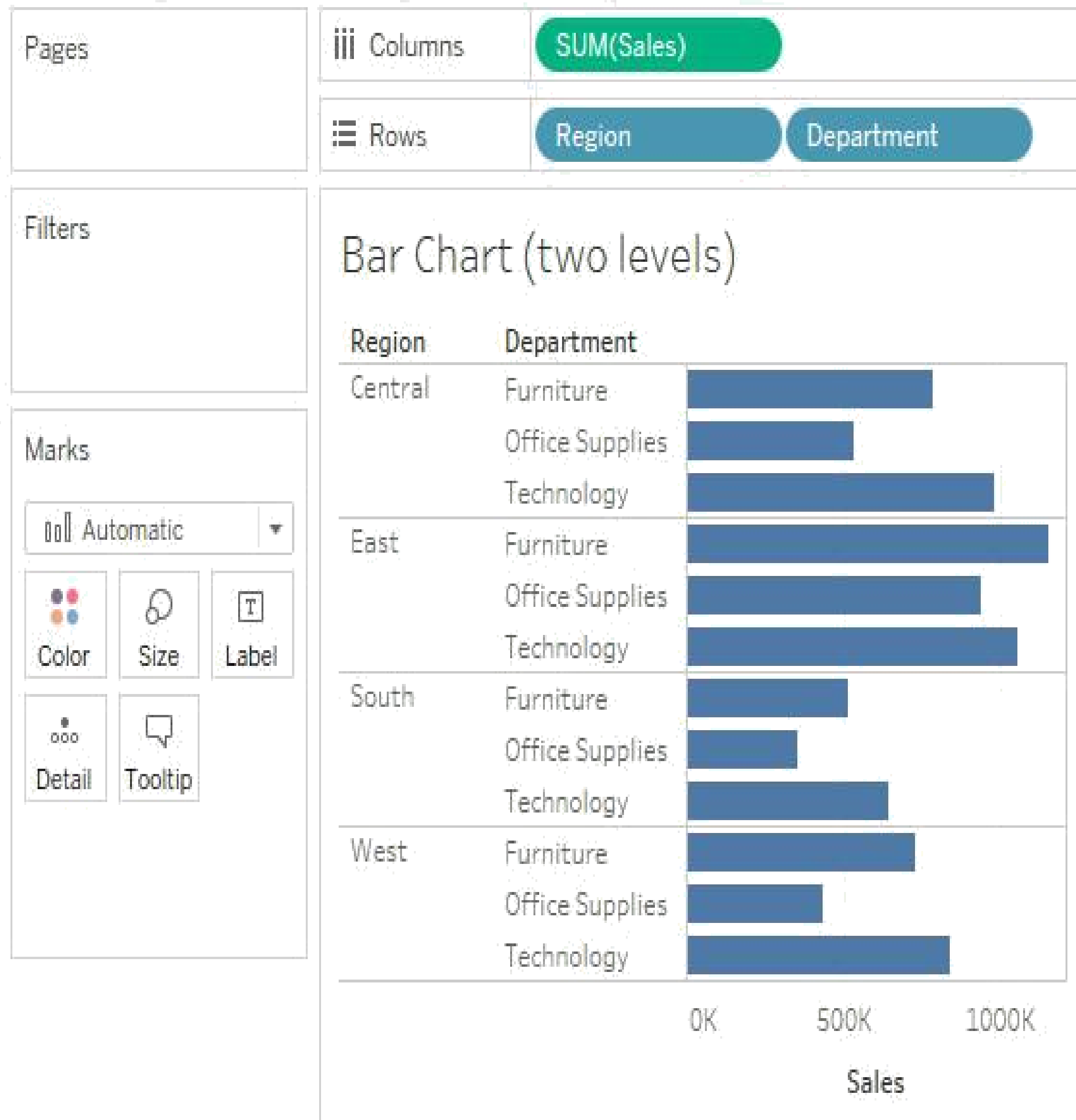
Circles and other shapes are drawn at locations defined by the value(s) of the field defining the axis. Take a moment to experiment with selecting different mark types from the drop-down menu on the Marks card. Having an understanding of how Tableau draws different mark types will help you master the tool.

**Iterations of bar charts for deeper analysis**

Using the preceding bar chart, you can easily see that the technology department has more total sales than either the furniture or office supplies departments. What if you want to further understand sales amounts for departments across various regions? Follow these two steps:

1. Navigate to the Bar Chart (two levels) sheet, where you will find an initial view identical to the one you created earlier
2. Drag the Region field from Dimensions in the data pane to the Rows shelf and drop it to the left of the Department field already in view

You should now have a view that looks like this:




You still have a horizontal bar chart, but now you've introduced Region as another dimension that changes the level of detail in the view and further slices the aggregate of the sum of sales. By placing Region before Department, you are able to easily compare the sales of each department within a given region.

Now you are starting to make some discoveries. For example: the Technology department has the most sales in every region, except in the East, where Furniture had higher sales. Office Supplies never has the highest sales in any region.

Let's take a look at a different view, using the same fields arranged differently:

1. Navigate to the Bar Chart (stacked) sheet, where you will find a view identical to the original bar chart.
2. Drag the Region field from the Rows shelf and drop it on to the Color shelf:

Instead of a **side-by-side** bar chart, you now have a **stacked bar**

chart. Each segment of the bar is color-coded by the Region field. Additionally, a color legend has been added to the workspace. You haven't changed the level of detail in the view, so sales are still summed for every combination of region and department:

*The* **View Level of Detail** *is a key concept when working with Tableau. In most basic visualizations, the combination of values of all the dimensions in the view defines the lowest level of detail for that view. All measures will be aggregated or sliced by the lowest level of detail. In the case of most simple views, the number of marks (indicated in the lower-left status bar) corresponds to the number of intersections of dimensional values. That is, there will be one mark for each combination of dimension values.*

If Department is the only field used as a dimension, you will have a view at the department level of detail, and all measures in the view will be aggregated per department.

If Region is the only field used as a dimension, you will have a view at the region level of detail, and all measures in the view will be aggregated per region.

If you use both Department and Region as dimensions in the view, you will have a view at the level of department and region. All measures will be aggregated per unique combination of department and region, and there will be one mark for each combination of department and region.

Stacked bars can be useful when you want to understand part-to-whole relationships. It is now fairly easy to see what portion of the total sales of each department is made in each region. However, it is very difficult to compare sales for most of the regions across departments. For example, can you easily tell which department had the highest sales in the East region? It is difficult because, with the exception of West, every segment of the bar has a different starting place.

Now take some time to experiment with the bar chart to see what variations you can create:

3. Navigate to the Bar Chart (experimentation) sheet.
4. Try dragging the Region field from Color to the other various shelves on the Marks card, such as Size, Label, and Detail. Observe that in each case the bars remain stacked but are redrawn based on the visual encoding defined by the Region field.

5. Use the **Swap** button on the Toolbar to swap fields on Rows and Columns. This allows you to very easily change from a horizontal bar chart to a vertical bar chart (and vice versa):

6. Drag and drop Sales from the Measures section of the data pane on top of the Region field on the Marks card to replace it. Drag the Sales field to Color if necessary, and notice how the color legend is a gradient for the continuous field.
7. Experiment further by dragging and dropping other fields onto various shelves. Note the behavior of Tableau for each action you take.
8. From the File menu, select Save.

*Tableau has an auto-save feature! If your machine crashes, then the next time you open Tableau, you will be prompted to open any previously-open workbooks that had not been saved. You should still develop a habit of saving your work early and often, though, and maintaining appropriate backups.*

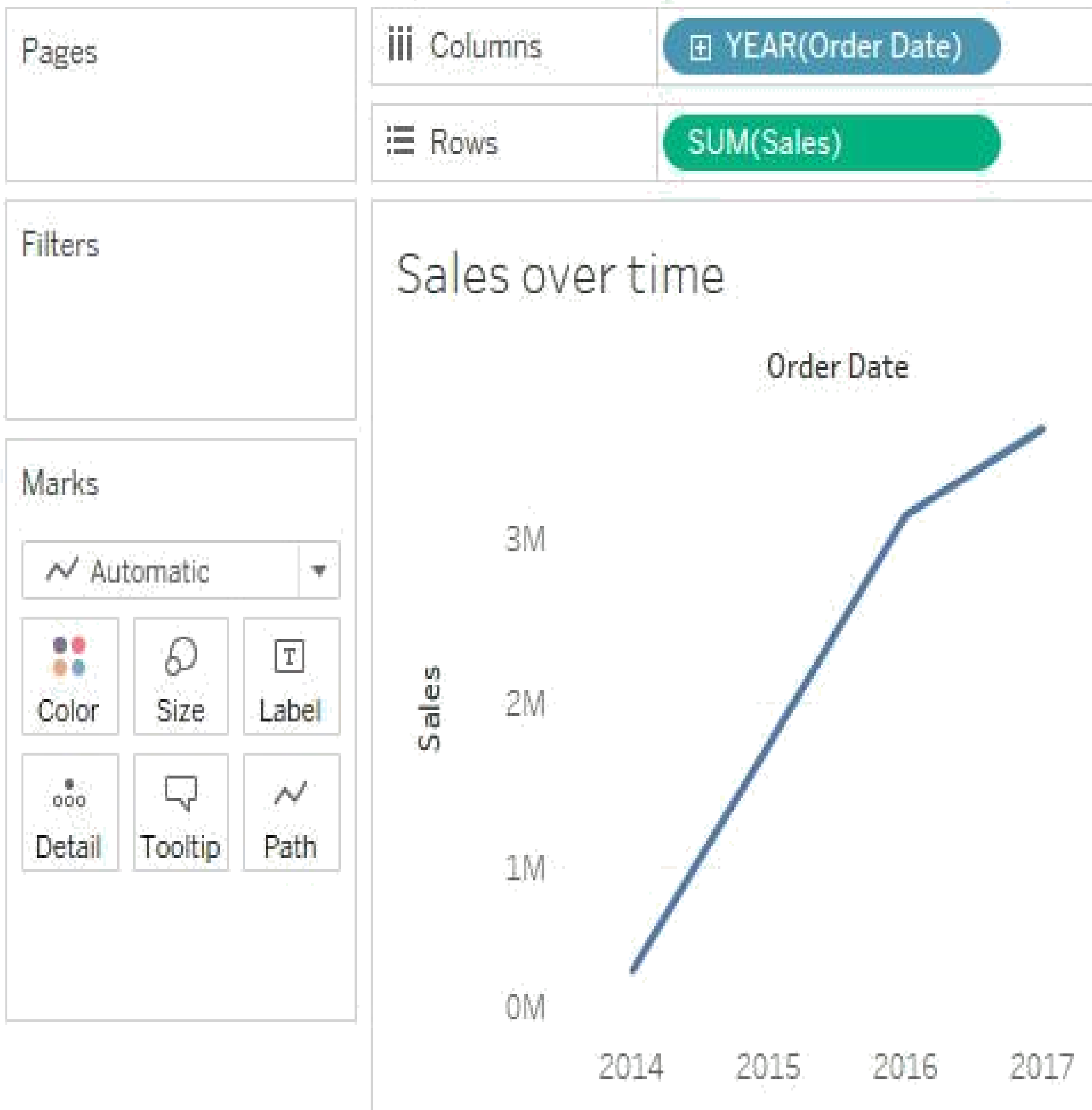
**Line charts**

**Line charts** connect related marks in a visualization to show movement or relationship between those connected marks. The position of the marks and the lines that connect them are the primary means of communicating the data. Additionally, you can use size and color to communicate additional information.

The most common kind of line chart is a **Time Series**. A time series shows the movement of values over time. Creating one in Tableau requires only a date and a measure.

Continue your analysis of Superstore sales using the Chapter 01 Starter workbook you just saved:

1. Navigate to the Sales over time sheet.
2. Drag the Sales field from Measures to Rows. This gives you a single, vertical bar representing the sum of all sales in the data source.
3. To turn this into a time series, you must introduce a date. Drag the Order Date field from Dimensions in the data pane on the left and drop it into Columns. Tableau has a built-in date hierarchy, and the default level of Year has given you a line chart connecting four years. Notice that you can clearly see an increase in sales year after year:


Pages
Columns
YEAR(Order Date)
Rows
SUM(Sales)
Filters
Sales over time
Order Date
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Path
Sales
3M
2M
1M
0M
2014
2015
2016
2017

4. Use the drop-down menu on the YEAR(Order Date) field on Columns (or right-click the field) and switch the date field to use Quarter. You may notice that Quarter is listed twice in the drop-down menu. We'll explore the various options for date parts, values, and hierarchies in the *Visualizing Dates and Times* section of Chapter 3, *Venturing on to Advanced Visualizations*. For now, select the second option

Columns
YEAR(Order Da..
Rows

Sales over ti

3500K
3000K
2500K

Filter...

Show Filter
Show Highlighter

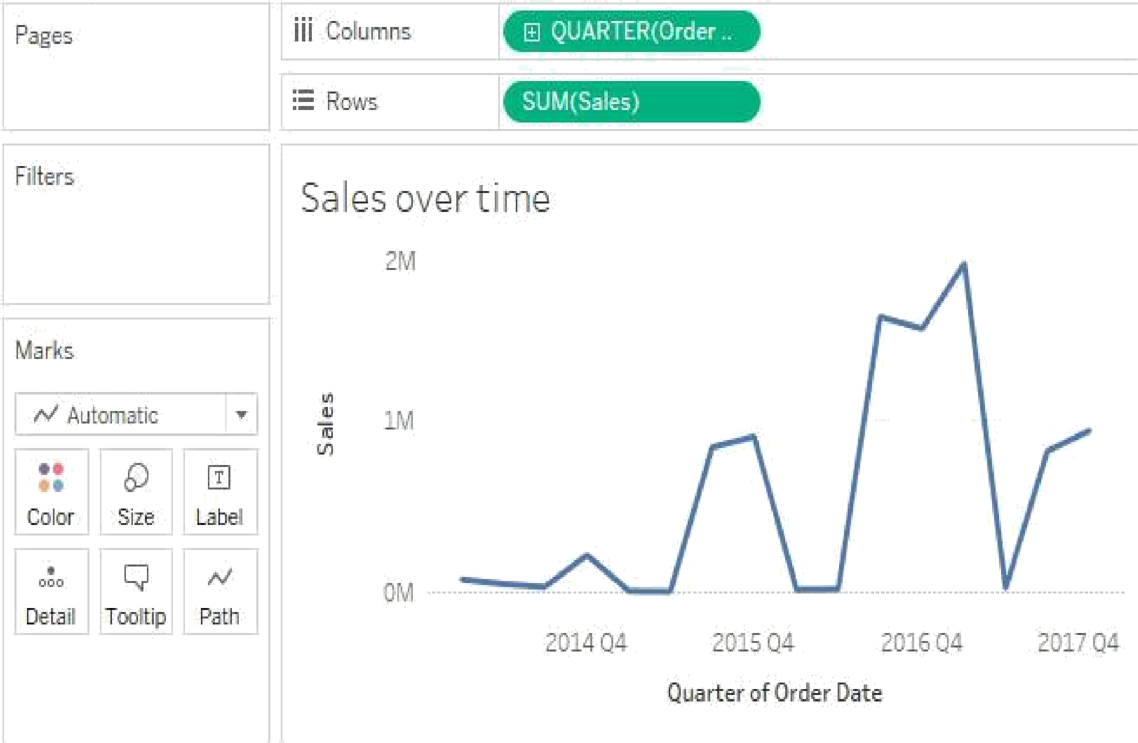
Sort...
Format...
✓ Show Header
✓ Include in Tooltip

Show Missing Values

| | |
|---|---|
| Year | 2015 |
| Quarter | Q2 |
| Month | May |
| Day | 8 |
| More | ▸ |

| | |
|---|---|
| Year | 2015 |
| Quarter | Q2 2015 |
| Month | May 2015 |
| Week Number | Week 5, 2015 |
| Day | May 8, 2015 |
| More | ▸ |

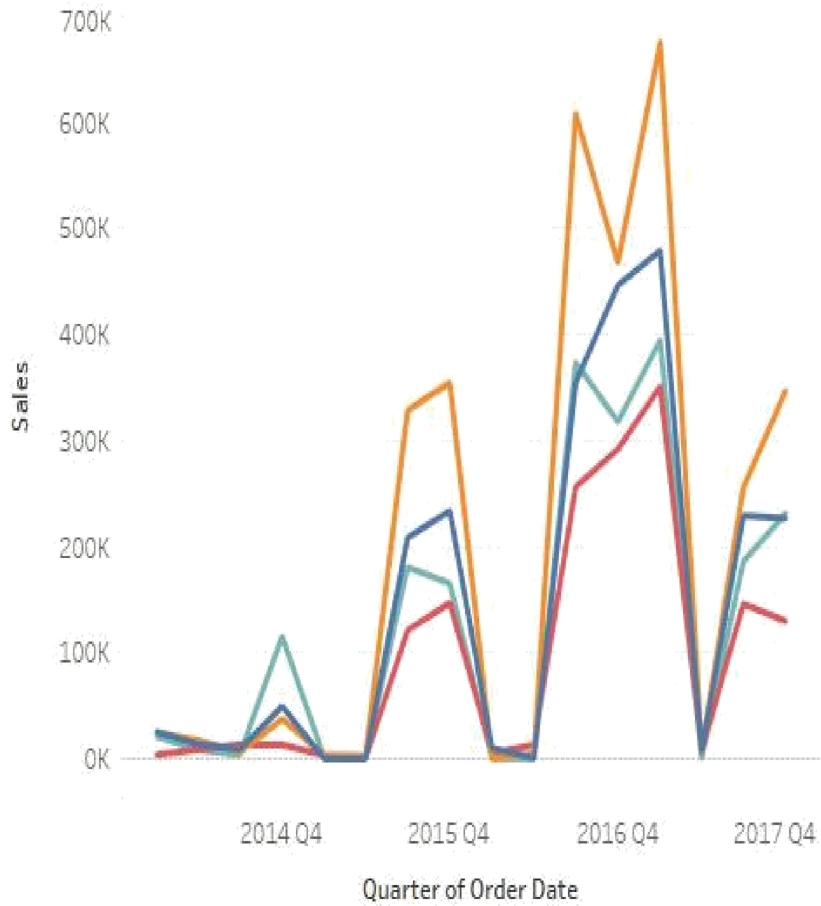
Notice that the cyclical pattern is quite evident when looking at sales by quarter:


Pages
Columns
QUARTER(Order ..
Rows
SUM(Sales)
Filters
Sales over time
2M
1M
0M
Sales
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Path
2014 Q4
2015 Q4
2016 Q4
2017 Q4
Quarter of Order Date

**Iterations of line charts for deeper analysis**

Right now, you are looking at the overall sales over time. Let's do some analysis at a slightly deeper level:

1. Navigate to the Sales over time (overlapping lines) sheet, where you will find a view identical to the one you just created.
2. Drag the Region field from Dimensions to Color. Now you have a line per region, with each line a different color, and a legend indicating which color is used for which region. As with the bars, adding a dimension to color splits the marks. However, unlike the bars, where the segments were stacked, the lines are not stacked. Instead, the lines are drawn at the exact value for the sum of sales for each region and quarter. This allows for easy and accurate comparison. It is interesting to note that the cyclical pattern can be observed for each region:



Pages
Columns
QUARTER(Order ..
Rows
SUM(Sales)
Filters
Sales over time (overlapping lines)
Marks
Automatic
Color
Size
Label
Detail
Tooltip
Path
Region
700K
600K
500K
400K
300K
200K
100K
0K
Sales
2014 Q4
2015 Q4
2016 Q4
2017 Q4
Quarter of Order Date
Region
Central
East
South
West

With only four regions, it's fairly easy to keep the lines separate. But what about dimensions that have even more distinct values? The steps are as follows:

1. Navigate to the Sales over time (multiple rows) sheet, where you will find a view identical to the one you just created.
2. Drag the Category field from Dimensions and drop it directly on top of the Region field currently on the Marks card. This replaces the

Region field with Category. You now have 17 overlapping lines. Often, you'll want to avoid more than two or three overlapping lines. But you might also consider using color or size to showcase an important line in the context of the others. Also note that clicking an item in the color legend will highlight the associated line in the view. Highlighting is an effective way to pick out a single item and compare it to all the others.

3. Drag the Category field from Color on the Marks card and drop it into Rows. You now have a line chart for each category. Now you have a way of comparing each product over time without an overwhelming overlap, and you can still compare trends and patterns over time. This is the start of a spark-lines visualization that will be developed more fully in the *Advanced Visualizations* section of Chapter 11, *Advanced Visualizations, Techniques, Tips, and Tricks*.

**Geographic visualizations**

In Tableau, the built-in geographic database recognizes geographic roles

for fields, such as Country, State, City, Airport, Congressional District, or Zip Code. Even if your data does not contain latitude and longitude values, you can simply use geographic fields to plot locations on a map. If your data does contain latitude and longitude fields, you may use those instead of the generated values.

*Tableau will automatically assign geographic roles to some fields based on a field name and a sampling of values in the data. You can assign or reassign geographic roles to any field by right-clicking the field in the data pane and using the Geographic Role option. This is also a good way to see what built-in geographic roles are available.*

Tableau can also read shape files and geometries from some databases. These and other geographic capabilities will be covered in more detail in the *Mapping Techniques* section of Chapter 11, *Advanced Visualizations, Techniques, Tips, and Tricks.* In the following examples, we'll consider some of the key concepts of geographic visualizing.

Geographic visualization is incredibly valuable when you need to understand where things happen and whether there are any spatial relationships within the data. Tableau offers three main types of geographic visualization:

- Filled maps (simply referred to as *maps* in the Tableau interface)
- Symbol maps

Density maps

**Filled maps**

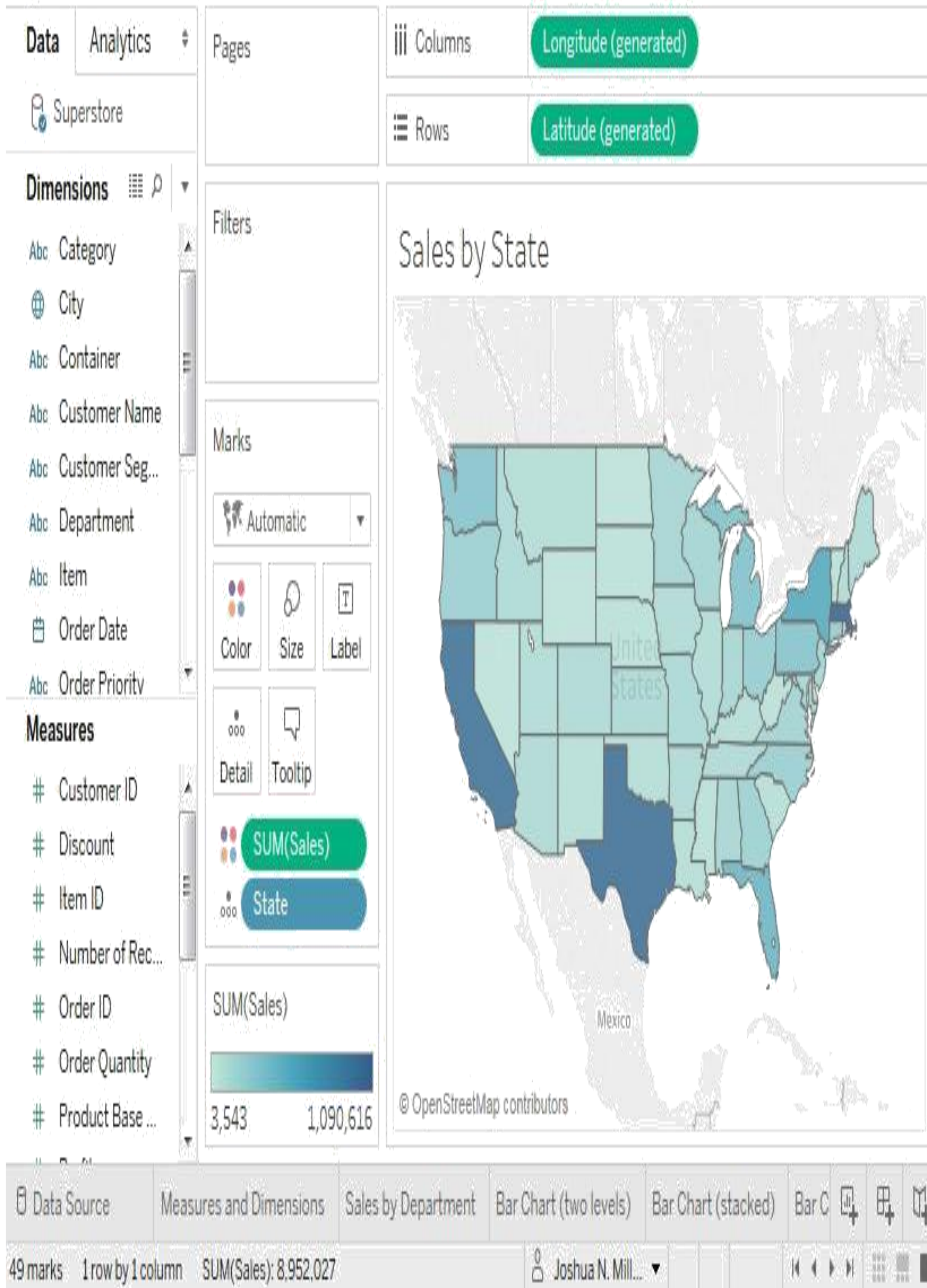
Filled maps fill areas such as countries, states, counties, or ZIP codes to show a location. The color that fills the area can be used to encode values, most often of aggregated measures but sometimes also dimensions. These maps are also called **choropleth maps**.

Let's say you want to understand sales for Superstore and see whether there are any patterns geographically. You might take an approach similar to the following:

1. Navigate to the Sales by State sheet.
2. Double-click the State field in the data pane. Tableau automatically creates a geographic visualization using the Latitude (generated),

   Longitude (generated), and State fields.

3. Drag the Sales field from the data pane and drop it on the Color shelf on the Marks card. Based on the fields and shelves you've used, Tableau has switched the automatic mark type to Map:



Data
Analytics
Superstore
Dimensions
Abc Category
City
Abc Container
Abc Customer Name
Abc Customer Seg...
Abc Department
Abc Item
Order Date
Abc Order Priority
Measures
Customer ID
Discount
Item ID
Number of Rec...
Order ID
Order Quantity
Product Base ...
Pages
Filters
Marks
Automatic
Color
Size
Label
Detail
Tooltip
SUM(Sales)
State
SUM(Sales)
3,543
1,090,616
Columns
Longitude (generated)
Rows
Latitude (generated)
Sales by State
United States
Mexico
© OpenStreetMap contributors
Data Source
Measures and Dimensions
Sales by Department
Bar Chart (two levels)
Bar Chart (stacked)
49 marks
1 row by 1 column
SUM(Sales): 8,952,027
Joshua N. Mill...

The filled map fills each state with a single color to indicate the relative sum of sales for that state. The color legend, now visible in the view, gives

the range of values and indicates that the state with the least sales had a total of **3,543** and and the state with the most sales had a total of **1,090,616**.

When you look at the number of marks displayed in the bottom status bar, you'll see that it is 49. Careful examination reveals that the marks consist of the lower 48 states and Washington DC; Hawaii and Alaska are not shown. Tableau will only draw a geographic mark, such as a filled state, if it exists in the data and is not excluded by a filter.

Observe that the map does display Canada, Mexico, and other locations not included in the data. These are part of a background image retrieved from an online map service. The state marks are then drawn on top of the background image. We'll look at how you can customize the map and even use other map services in the *Mapping Techniques* section of Chapter 11, *Advanced Visualizations, Techniques, Tips, and Tricks*.

Filled maps can work well in interactive dashboards and have quite a bit of aesthetic value. However, certain kinds of analyses are very difficult with filled maps. Unlike other visualization types, where size can be used to communicate facets of the data, the size of a filled geographic region only relates to the geographic size and can make comparisons difficult. For example: which state has the highest sales? You might be tempted to say Texas or California because they appear larger, but would you have guessed Massachusetts? Some locations may be small enough that they won't even show up compared to larger areas. Use filled maps with caution and consider pairing them with other visualizations on dashboards for clear communication.

**Symbol maps**

With symbol maps, marks on the map are not drawn as filled regions; rather, marks are shapes or symbols placed at specific geographic locations. The size, color, and shape may also be used to encode additional dimensions and measures.

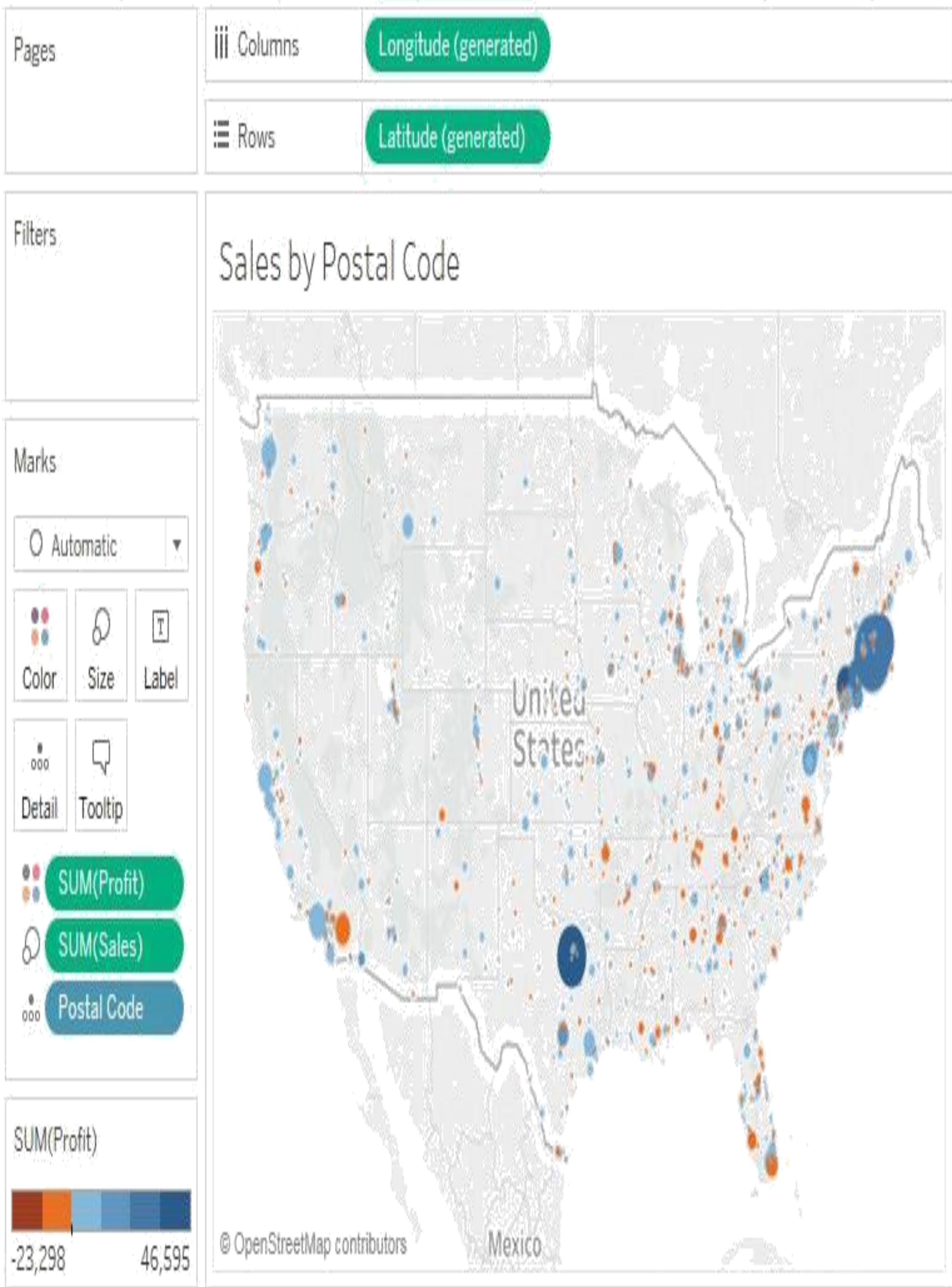
Continue your analysis of Superstore sales by following these steps:

1. Navigate to the Sales by Postal Code sheet.
2. Double-click Postal Code under Dimensions. Tableau automatically adds Postal Code to the Detail of the Marks card and Longitude

   (generated) and Latitude (generated) to Columns and Rows. The mark

   type is set to a circle by default, and a single circle is drawn for each postal code at the correct latitude and longitude.
3. Drag Sales from Measures to the Size shelf on the Marks card. This causes each circle to be sized according to the sum of sales for that postal code.
4. Drag Profit from Measures to the Color shelf on the Marks card. This encodes the mark color to correspond to the sum of profit. You can now see the geographic location of profit and sales at the same time. This is useful because you will see some locations with high sales and low profit, which may require some action.

The final view should look like this, after making some fine-tuned adjustments to the size and color:


Pages
Columns
Longitude (generated)
Rows
Latitude (generated)
Filters
Sales by Postal Code
Marks
Automatic
Color
Size
Label
Detail
Tooltip
SUM(Profit)
SUM(Sales)
Postal Code
SUM(Profit)
-23,298
46,595
United States
© OpenStreetMap contributors
Mexico

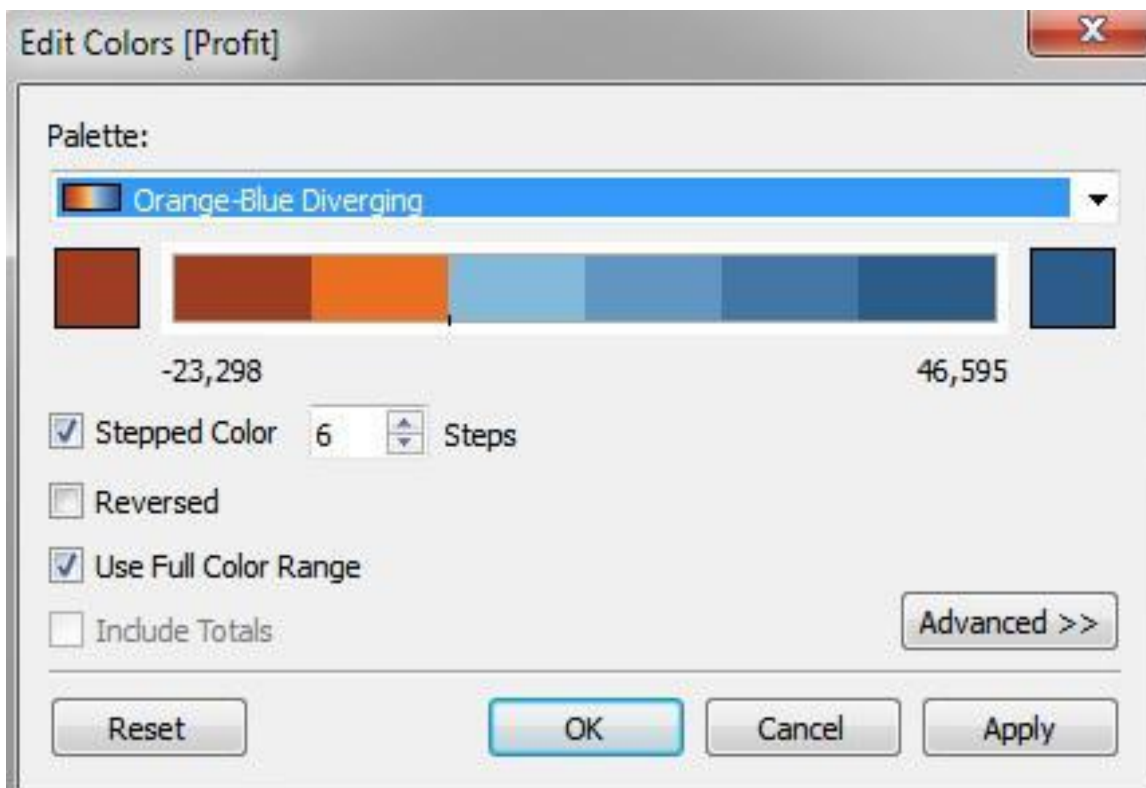
Sometimes, you'll want to adjust the marks on symbol map to make them more visible. Some options include the following

- If the marks are overlapping, click the Color shelf and set the transparency to somewhere between 50% and 75%. Additionally, add a dark border. This makes the marks stand out, and you can often better discern any overlapping marks.

- If marks are too small, click on the Size shelf and adjust the slider. You may also double-click the size legend and edit the details of how Tableau assigns size.

- If the marks are too faint, double-click the color legend and edit the details of how Tableau assigns color. This is especially useful when you are using a continuous field that defines a color gradient.

A combination of tweaking the size and using Stepped Color and Use Full Color Range, as shown here, produced the final result for this example:



Unlike filled maps, symbol maps allow you to use size to visually encode aspects of the data. Symbol maps also allow for greater precision. In fact, if you have latitude and longitude in your data, you can very precisely plot marks at a street address-level of detail. This type of visualization also allows you to map locations that do not have clearly defined boundaries.

Sometimes, when you manually select **Map** in the **Marks** card drop-down menu, you will get an error message indicating that filled maps are not supported at the level of detail in the view. In those cases, Tableau is rendering a geographic location that does not have built-in shapes. Other than cases where filled maps are not possible, you will need to decide which type best meets your needs. We'll also consider the possibility of combining filled maps and symbol maps in a single view in later chapters.

**Density maps**

Density maps show the spread and concentration of values within a geographic area. Instead of individual points or symbols, the marks blend together, showing intensity in areas with a high concentration. You can control color, size, and intensity.

Let's say you want to understand the geographic concentration of orders.

You might create a density map using the following steps:

1. Navigate to the Density of Orders sheet.
2. Double-click the Postal Code field in the data pane. Just as before, Tableau automatically creates a symbol map geographic

   visualization using the Latitude (generated), Longitude (generated),

   and State fields.
3. Using the drop-down menu on the Marks card, change the mark type to Density. The individual circles now blend together showing concentrations:

Pages

Columns: Longitude (generated)

Rows: Latitude (generated)

Filters

Marks

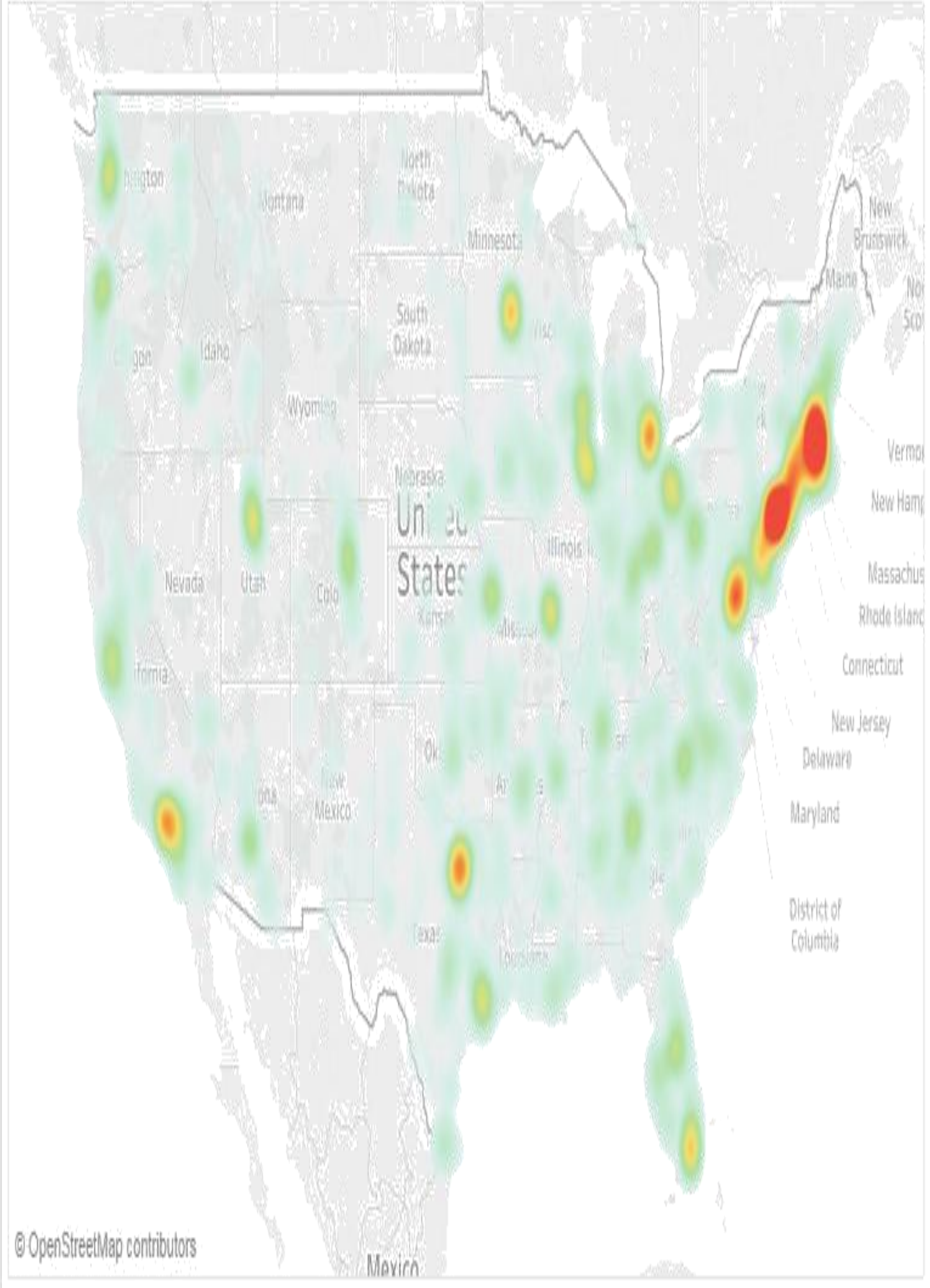
Density

Color | Size | Label

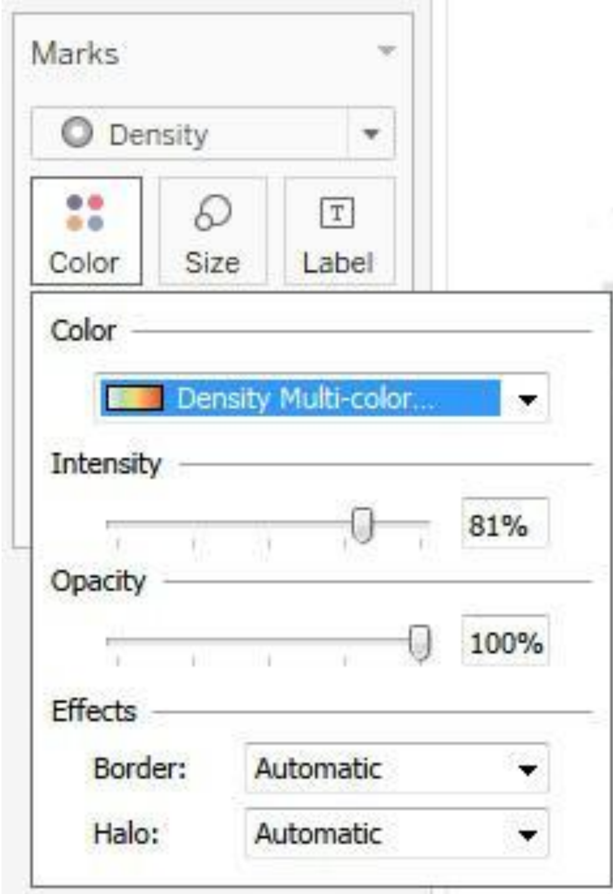
Detail | Tooltip

Postal Code

Density of Orders



Try experimenting with the Color and Size options. Clicking on Color, for example, reveals some options specific to the Density mark type:



Several color palettes are available that work well for density marks (the default ones work well with light color backgrounds, but there are others designed to work with dark color backgrounds). The Intensity slider allows you to determine how intense the marks should be drawn based on concentrations.

This density map displays a high concentration of orders from the East Coast. Sometimes, you'll see patterns that merely reflect population density—in which case, your analysis may not be particularly meaningful. In this case, the concentration on the East Coast compared to the lack of density on the west coast is intriguing.

**Using Show Me**

Show Me is a powerful component of Tableau that arranges selected and active fields into the places required for the selected visualization type. The Show Me toolbar displays small thumbnail images of different types of visualizations, allowing you to create visualizations with a single click. Based on the fields you select in the data pane and the fields that are already in view, Show Me will enable possible visualizations and highlight a recommended visualization.

Explore the features of Show Me by following these steps:

1. Navigate to the Show Me sheet.
2. If the Show Me pane is not expanded, click the Show Me button in the upper-right of the toolbar to expand the pane.
3. Press and hold the *Ctrl* key while clicking the Postal Code, State, and Profit fields in the data pane to select each of those fields. With those fields highlighted, Show Me should look like this:

Notice that the Show Me window has enabled certain visualization types

such as **text tables**, **heat maps**, **symbol maps**, **filled maps**, and **bar charts**. These are the visualizations that are possible given the fields already in the view, in addition to any selected in the data pane. Show Me highlights the recommended visualization for the selected fields and also gives a description of what fields are required as you hover over each visualization type. Symbol maps, for example, require one geographic dimension and 0 to 2 measures.

Other visualizations are greyed-out, such as lines, area charts, and histograms. Show Me will not create these visualization types with the fields that are currently in the view and selected in the data pane. Hover over the greyed-out line-charts option in Show Me. It indicates that line charts require one or more measures (which you have selected) but also require a date field (which you have not selected).

*Tableau will draw line charts with fields other than dates. Show Me gives you options for what is typically considered good practice for visualizations. However, there may be times when you know that a line chart would represent your data better.*

*Understanding how Tableau renders visualizations based on fields and shelves instead of always relying on Show Me will give you much greater flexibility in your visualizations and will allow you to rearrange things when Show Me doesn't give the exact results you want. At the same time, you will need to cultivate an awareness of good visualization practices.*

Show Me can be a powerful way to quickly iterate through different visualization types as you search for insights into the data. But as a data explorer, analyst, and story-teller, you should consider Show Me as a helpful guide that gives suggestions. You may know that a certain visualization type will answer your questions more effectively than the suggestions of Show Me. You also may have a plan for a visualization type that will work well as part of a dashboard but isn't even included in Show Me.

You will be well on your way to learning and mastering Tableau when you can use Show Me effectively, but feel just as comfortable building visualizations without it. Show Me is powerful for quickly iterating through visualizations as you look for insights and raise new questions. It is useful for starting with a standard visualization that you will further customize. It is wonderful as a teaching and learning tool.

However, be careful not to use it as a crutch without understanding how visualizations are actually built from the data. Take time to evaluate why certain visualizations are or are not possible. Pause to see what fields and shelves were used when you selected a certain visualization type.

End the Show Me example by experimenting with Show Me by clicking various visualization types, looking for insights into the data that may be more or less obvious based on the visualization type. **Circle views** and **box-and-whisker plots** show the distribution of postal codes for each state. Bar charts easily expose several postal codes with negative profit.

Now that you have become familiar with creating individual views of the data, let's turn our attention to putting it all together in a dashboard.

**Putting everything together in a dashboard**

Often, you'll need more than a single visualization to communicate the full story of the data. In these cases, Tableau makes it very easy for you to use multiple visualizations together on a dashboard. In Tableau, a *dashboard* is a collection of views, filters, parameters, images, and other objects that work together to communicate a data story. Dashboards are often interactive and allow end users to explore different facets of the data.

Dashboards serve a wide variety of purposes and can be tailored to suit a wide variety of audiences. Consider the following possible dashboards:

- A summary-level view of profit and sales allows executives to have a quick glimpse into the current status of the company
- An interactive dashboard, allowing sales managers to drill into sales territories to identify threats or opportunities
- A dashboard allowing doctors to track patient readmissions, diagnoses, and procedures to make better decisions about patient care

A dashboard allowing executives of a real-estate company to identify trends and make decisions for various apartment complexes

An interactive dashboard for loan officers to make lending decisions based on portfolios broken down by credit ratings and geographic location

Considerations for different audiences and advanced techniques will be covered in detail in Chapter 7, *Telling a Data Story with Dashboards*.

**The Dashboard interface**

When you create a new dashboard, the interface will be slightly different than it is when designing a single view. We'll start designing your first dashboard after a brief look at the interface. You might navigate to the Superstore Sales sheet and take a quick look at it yourself.

The dashboard window consists of several key components. Techniques for using these objects will be detailed in Chapter 7*, Telling a Data Story with Dashboards.* For now, focus on gaining some familiarity with the options that are available. One thing you'll notice is that the left sidebar has been replaced with dashboard-specific content:

The left side-bar contains two tabs:

- A Dashboard tab, for sizing options and adding sheets and objects

to the dashboard

A Layout tab, for adjusting the layout of various objects on the dashboard

The Dashboard pane contains options for previewing based on target device along with several sections:

A **Size** section, for dashboard sizing options

A **Sheets** section, containing all sheets (views) available to place on the dashboard

An **Objects** section with additional objects that can be added to the dashboard

You can add sheets and objects to a dashboard by dragging and dropping. As you drag the view, a light-grey shading will indicate the location of the sheet in the dashboard once it is dropped. You can also double-click any sheet and it will be added automatically.

In addition to adding sheets, the following objects may be added to the dashboard:

**Horizontal** and **Vertical** layout containers will give you finer control over the layout

**Text** allows you to add text labels and titles

An **Image** and even embedded **Web Page** content can be added

A **Blank** object allows you to preserve blank space in a dashboard, or it can serve as a place holder until additional content is designed

- A **Button** is an object that allows the user to navigate between dashboards

- An **Extension** gives you the ability to add controls and objects

that you or a third party have developed for interacting with the

dashboard and providing extended functionality

Using the toggle, you can select whether new objects will be added as Tiled or Floating. Tiled objects will snap into a tiled layout next to other tiled objects or within layout containers. Floating objects will float on top of the dashboard in successive layers.

When a worksheet is first added to a dashboard, any legends, filters, or parameters that were visible in the worksheet view will be added to the dashboard. If you wish to add them at a later point, select the sheet in the dashboard and click the little drop-down caret on the upper right. Nearly every object has the drop-down caret, providing many options for fine-tuning the appearance and controlling behavior.

Take note of the various **User Interface** (**UI**) elements that become visible for selected objects on the dashboard:

Grip

Remove from dashboard

Navigate to the sheet

Use the sheet as a filter on the dashboard

Open the drop down menu

Sales by Postal Code

United States

Mexico

Go to Sheet
Duplicate Sheet
Title
Caption
Legends
Filters
Highlighters
Show Page Control
View Toolbar
Use as Filter
Ignore Actions
Floating
Select Layout Container
Deselect
Remove from Dashboard

Color Legend (Profit)
Shape Legend
Size Legend (Sales)
Map Legend

You can resize an object on the dashboard using the border. The **Grip**, marked in the screenshot, allows you move the object once it has been placed. We'll consider other options as we go.

**Building your dashboard**

With an overview of the interface, you are now ready to build a dashboard by following these steps:

1. Navigate to the Superstore Sales sheet. You should see a blank dashboard.
2. Successively double-click each of the following sheets listed in the Dashboard section on the left: Sales by Department, Sales over time, and Sales by Postal Code. Notice that double-clicking the object adds it to the layout of the dashboard.
3. Add a title to the dashboard by checking Show Dashboard title in the lower left of the sidebar.
4. Select the Sales by Department sheet in the dashboard and click the drop-down arrow to show the menu.
5. Select Fit | Entire View. The **Fit** options describe how the visualization should fill any available space.

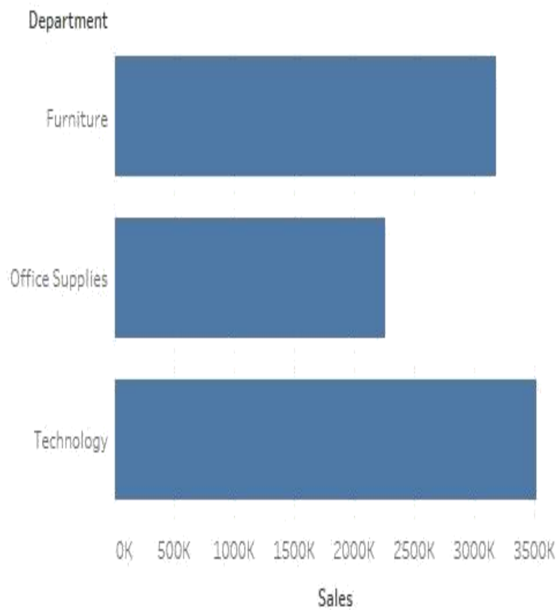
*Be careful when using various* **Fit** *options. If you are using a dashboard with a size that has not been fixed or if your view dynamically changes the number of items displayed based on interactivity, then what might have once looked good might not fit the view nearly as well.*

6. Select the Sales size legend by clicking it. Use the **Remove UI** element to remove the legend from the dashboard.
7. Select the Profit color legend by clicking it. Use the grip to drag the legend, and place it under the map.

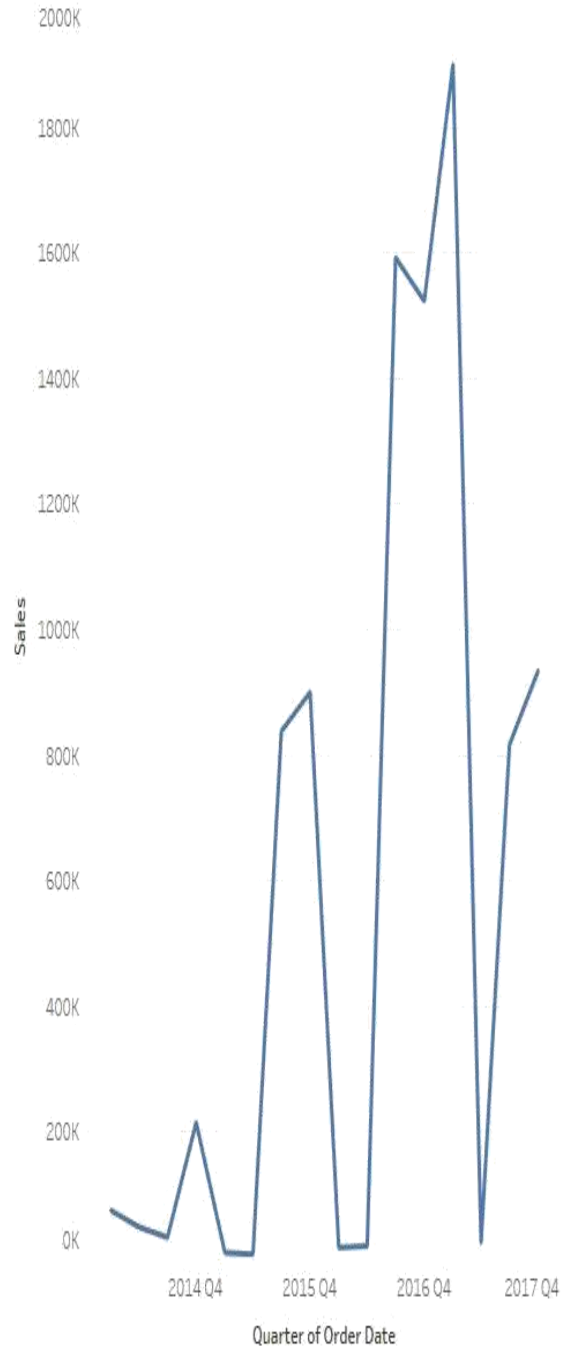
8. For each view (Sales by Department, Sales by Postal Code, and Sales over time), select the view by clicking an empty area in the view. Then, click the **Use as Filter UI** element to make that view an inte

Superstore Sales

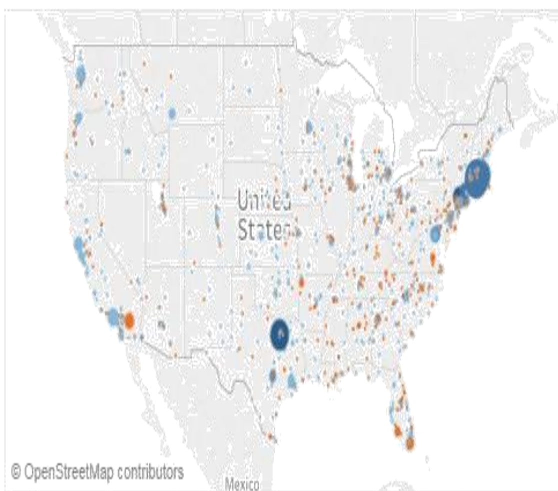
Sales by Department




Sales over time




Sales by Postal Code



9. Take a moment to interact with your dashboard. Click on various marks, such as the bars, states, and points of the line. Notice that each selection filters the rest of the dashboard. Clicking a selected mark will deselect it and clear the filter. Notice also that selecting marks in multiple views causes filters to work together. For example, selecting the bar for Furniture in Sales by Department and the 2016 Q1 in Sales over time allows you to see all the ZIP codes that had furniture sales in the first quarter of 2016.

Congratulations! You have now created a dashboard that allows for interactive analysis!

As an analyst for the Superstore chain, your visualizations allowed you to explore and analyze the data. The dashboard you created can be shared with members of management, and it can be used as a tool to help them see and understand the data to make better decisions. When a manager selects the furniture department, it immediately becomes obvious that there are locations where sales are quite high but the profit is actually very low. This may lead to decisions such as a change in marketing or a new sales focus for that location. Most likely, this will require additional analysis to determine the best course of action. In this case, Tableau will empower you to continue the cycle of discovery, analysis, and storytelling.

## Summary

Tableau's visual environment allows for a rapid and iterative process of exploring and analyzing data visually. You've taken your first steps toward understanding how to use the platform. You connected to data and then explored and analyzed the data using some key visualization types such as bar charts, line charts, and geographic visualizations. Along the way, you focused on learning the techniques and understanding key concepts such as the difference between measures and dimensions, and discrete and continuous fields. Finally, you put all the pieces together to create a fully functional dashboard that allows an end user to understand your analysis and make discoveries of their own.

In the next chapter, we'll explore how Tableau works with data. You will be exposed to fundamental concepts and practical examples of how to connect to various data sources. Combined with the key

concepts you just learned about building visualizations, you will be well equipped to move on to more advanced visualizations, deeper analysis, and telling fully interactive data stories.

## Working with Data in Tableau

Tableau offers the ability to connect to nearly any data source. It does so using a unique paradigm that allows it to leverage the power and efficiency of existing database engines with an option to extract data locally. This chapter focuses on essential concepts of how Tableau works with data, including the following topics:

- The Tableau paradigm
- Connecting to data
- Working with extracts instead of live connections
- Tableau file types
- Metadata and sharing connections
- Joins and blends
- Filtering data

**The Tableau paradigm**

The unique and exciting experience of working with data in Tableau is a result of (**VizQL Visual Query Language**).

VizQL was developed as a Stanford research project, focusing on the natural ways that humans visually perceive the world and how that could be applied to data visualization. We naturally perceive differences in size, shape, spatial location, and color. VizQL allows Tableau to translate your actions, as you drag and drop fields of data in a visual environment, into a query language that defines how the data encodes those visual elements. You will never need to read, write, or debug VizQL. As you drag and drop fields onto various shelves defining size, color, shape, and spatial location, Tableau will generate the VizQL behind the scenes. This allows you to focus on visualizing data, not writing code!

One of the benefits of VizQL is that it provides a common way of describing how the arrangement of various fields in a view defines a query related to the data. This common baseline can then be translated into numerous flavors of SQL, MDX, and **TQL** (short for **Tableau Query Language**, used for extracted data). Tableau will automatically perform the translation of VizQL into a native query to be run by the source data engine.

In its simplest form, the Tableau paradigm of working with data looks like the following diagram:

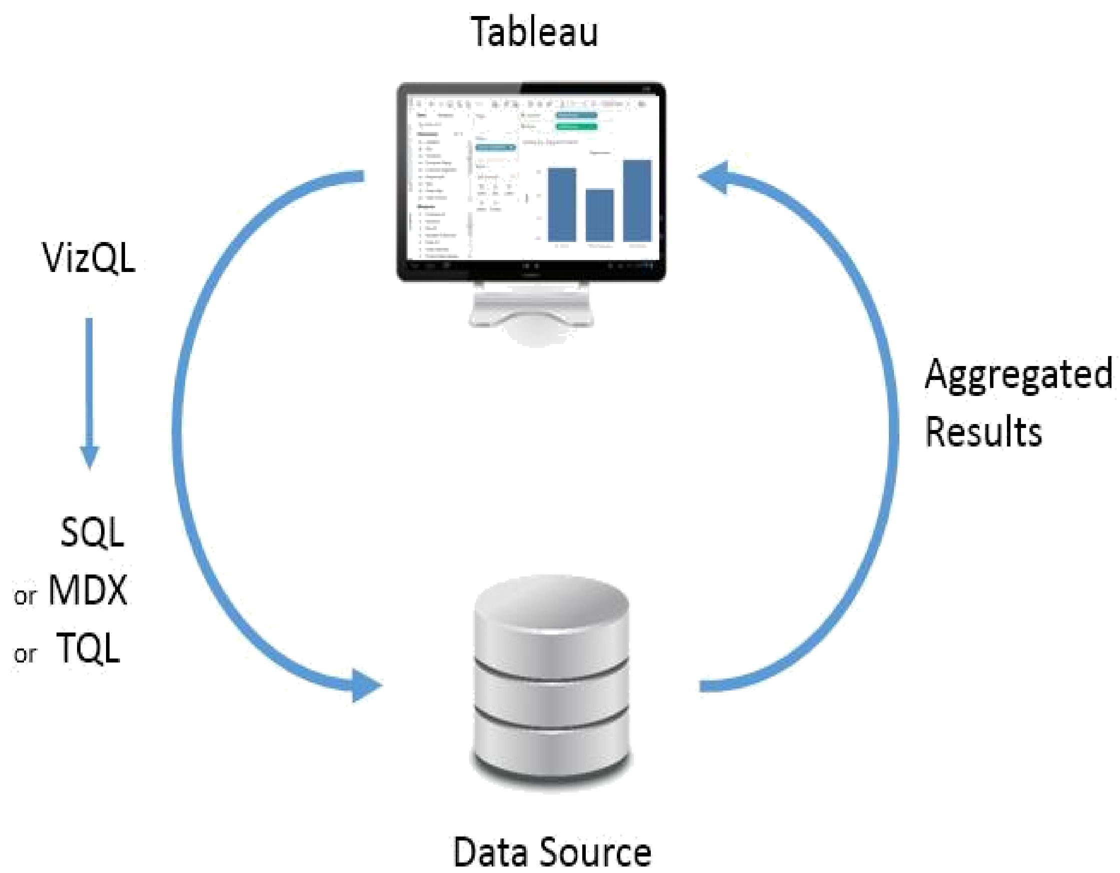
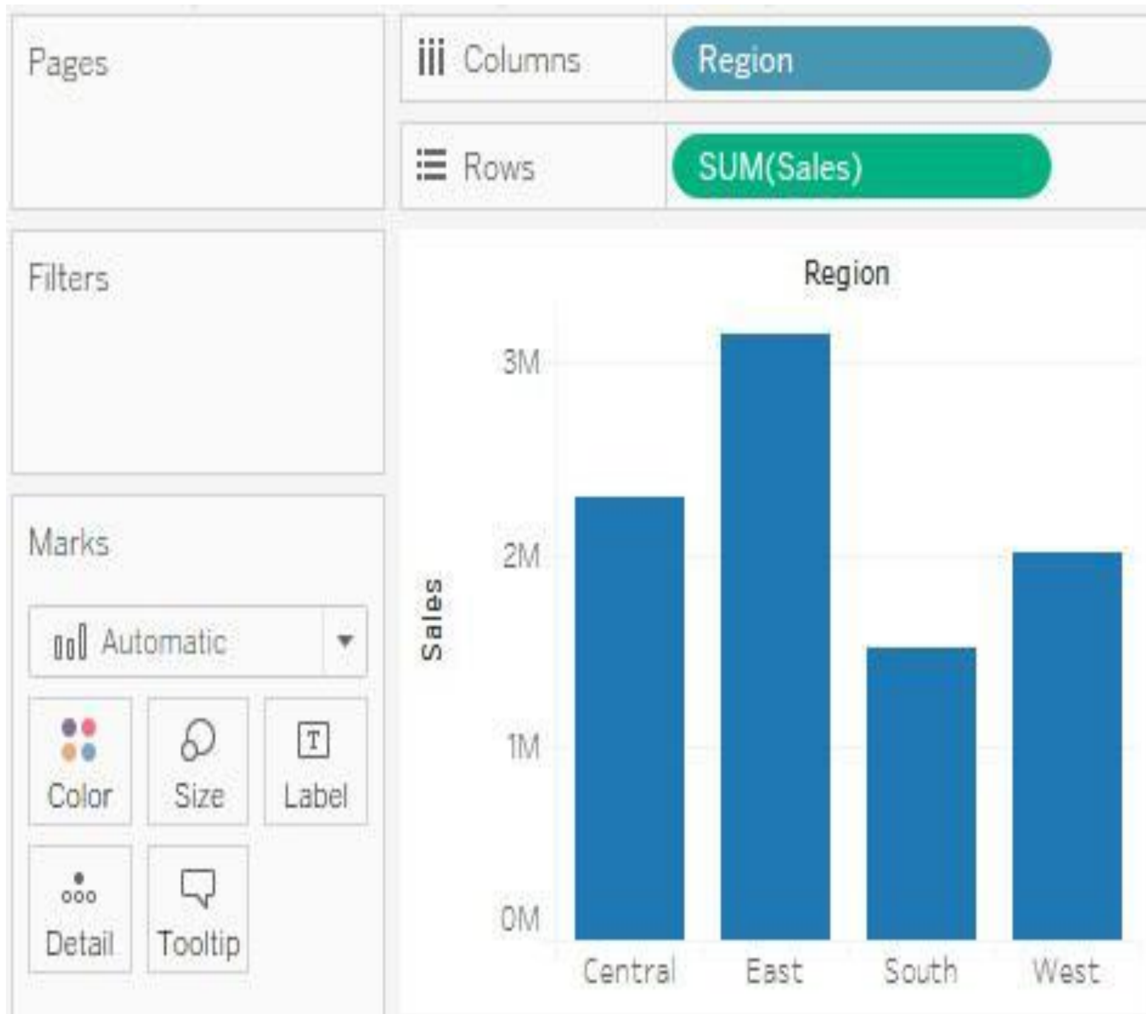


Tableau
VizQL
SQL
or MDX
or TQL
Aggregated
Results
Data Source

**A simple example**

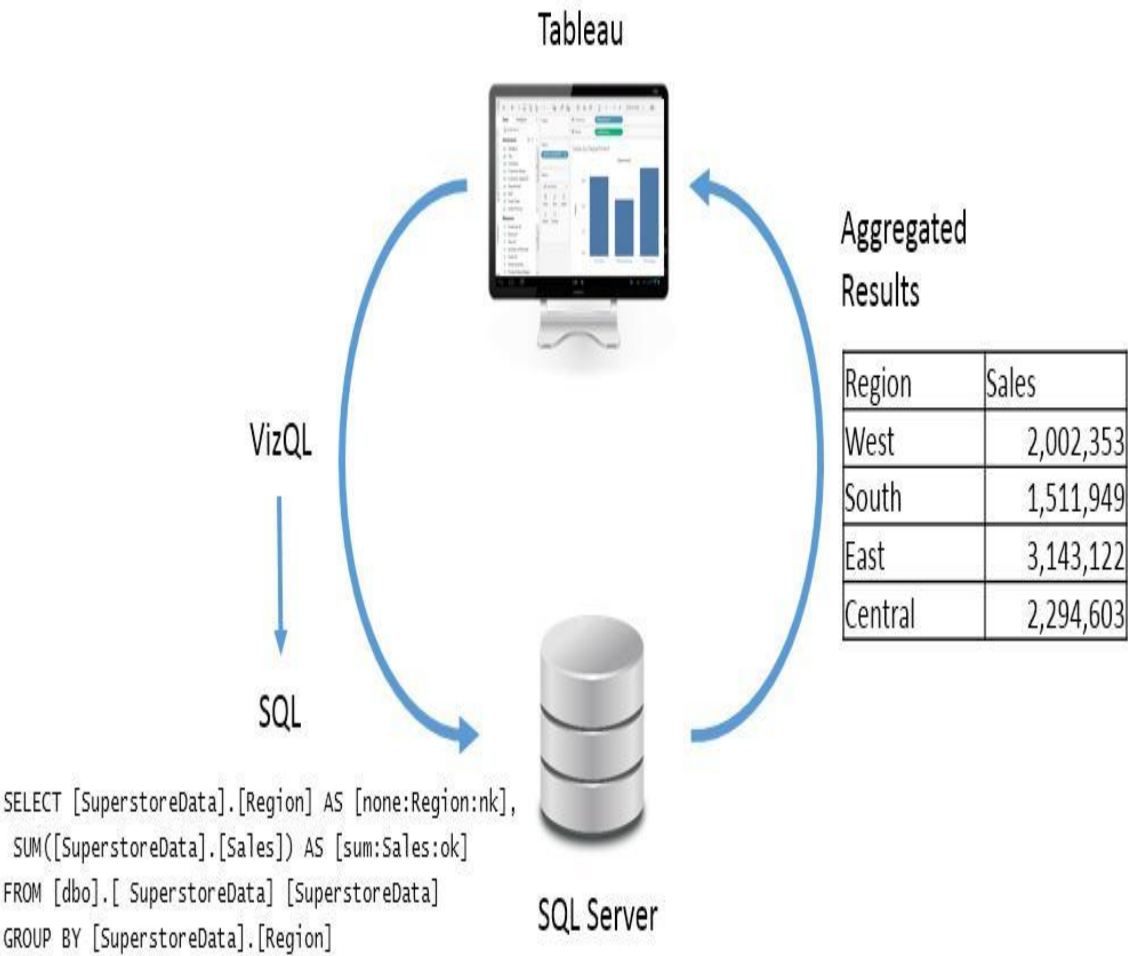
Go ahead and open the Chapter 02 Starter.twbx workbook located in the \Learning Tableau\Chapter 02 directory and navigate to the Tableau Paradigm sheet. Take a look at the following screenshot, which was created by dropping the Region dimension on Columns and the Sales measure on Rows:




The Region field is used as a discrete (blue) field in the view, and so defines column headers. As a dimension, it defines the level of detail in the view and slices the measure such that you get one bar per region. The Sales field

is a measure aggregated by summing each sale within each region. As a continuous (green) field, Sales defines an axis.

For this example (although the principle applies to any data source), let's say you were connected live to a SQL Server database with the Superstore data stored in a table. When you first create the preceding screenshot, Tableau generates a VizQL script, which is translated into SQL script and sent to the SQL Server. The SQL Server database engine evaluates the query and returns aggregated results to Tableau, which are then rendered visually. The entire process would look something like the following diagram in Tableau's paradigm:



There may have been hundreds, thousands, or even millions of rows of sales data in SQL Server. However, when SQL Server processes the query it returns aggregate results. In this case, SQL Server returns only four

aggregate rows of data to Tableau—one row for each region.

To see the aggregate data that Tableau used to draw the view, press *Ctrl + A* to select all the bars, and then right-click one of them and select View Data.

*On occasion, a database administrator may want to find out what scripts are running against a certain database to debug performance issues, or to determine more efficient indexing or data structures. Many databases supply profiling utilities or log execution of queries. In addition, you can find SQL or MDX generated by Tableau in the logs located in the My Tableau RepositoryLogs directory.*

*You may also use Tableau's built-in* **Performance Recorder** *to locate the queries that have been executed. From the top menu, select Help | Settings and Performance | Start Performance Recording, then interact with a view, and, finally, stop the recording from the menu. Tableau will open a dashboard that will allow you to see tasks, performance, and queries that were executed during the recording session.*

You can actually see the aggregate data that Tableau used to render the bars and the underlying records by following these steps:

1. Navigate to the Tableau Paradigm sheet in the Chapter 02 Starter workbook.
2. Press *Ctrl + A* to select all four bars.
3. Right-click one of the bars and select View Data... from the context menu as follows:

View Data

Show aliases Copy Export All

| Region | Sales |
|---|---|
| West | 2,002,353 |
| South | 1,511,949 |
| East | 3,143,122 |
| Central | 2,294,603 |

Summary Underlying 4 rows

The View Data screen allows you to observe the data in the view. The Summary tab displays the aggregate-level data that was used to render the view. The Sales values here are the sum of sales for each region. When you click the Underlying tab, Tableau will query the data source to retrieve all the records that make up the aggregate records. In this case, there are 9,426 underlying records, as indicated on the status bar in the lower-right corner of the following screenshot:

View Data

9,426 rows Show aliases Show all fields Copy Export All

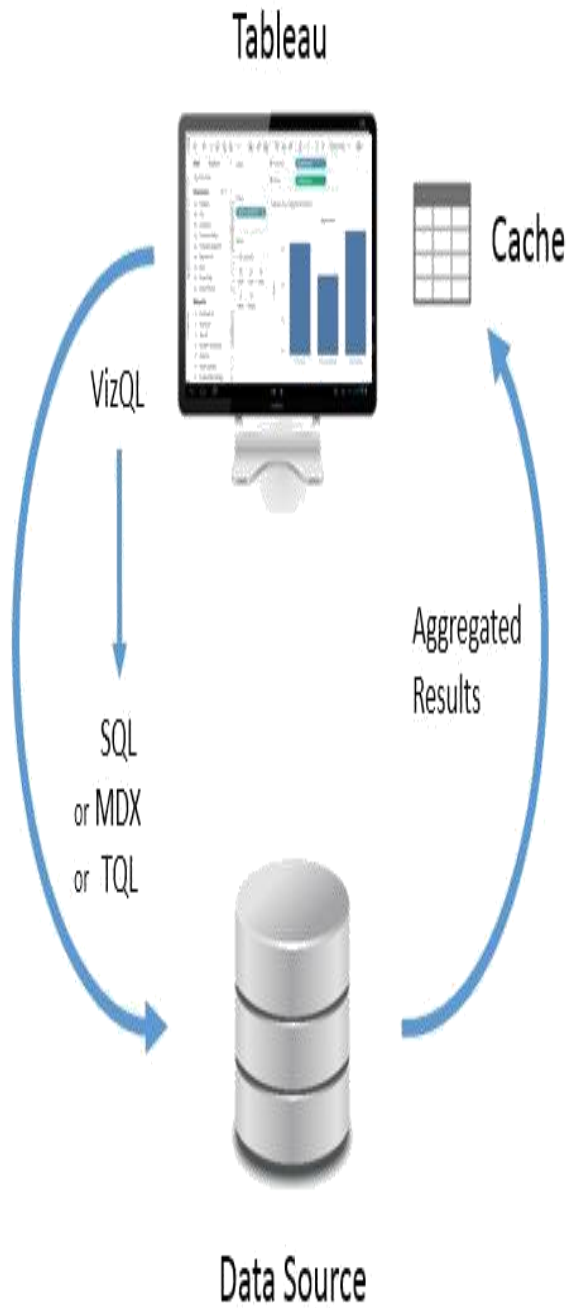
| Category | City | Container | Customer ID | Customer Name |
|---|---|---|---|---|
| Appliances | Lancaster | Small Box | 1890 | Jonathan Drummey |
| Paper | Oxford | Small Box | 607 | Joe Mako |
| Paper | Boston | Small Box | 608 | Chuck Hooper |
| Paper | Conway | Small Box | 3073 | Shawn Wallwork |

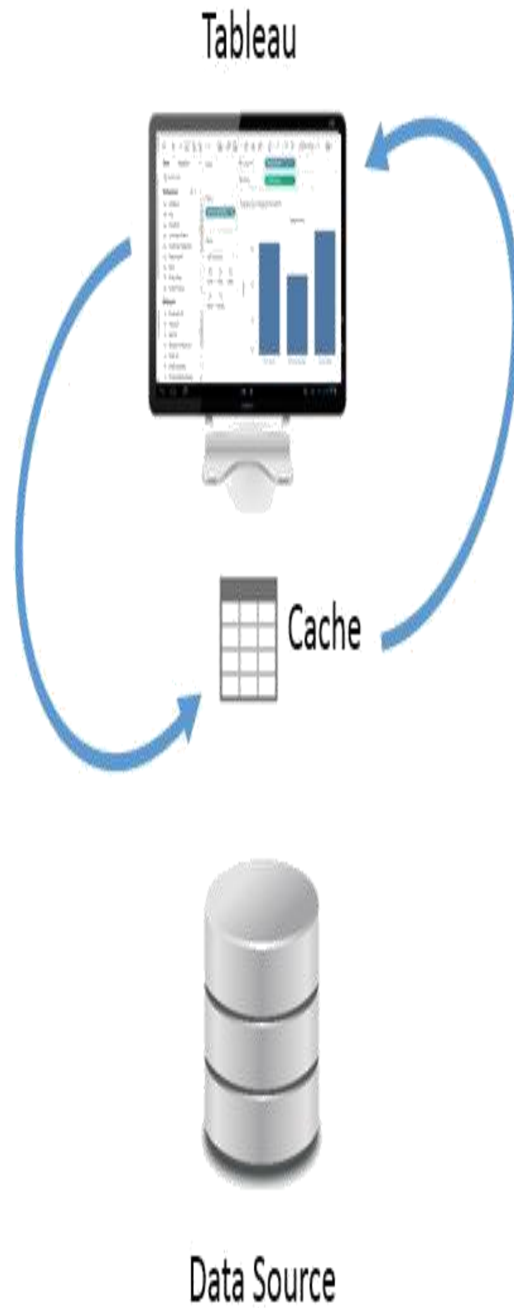
Summary Underlying 9,426 rows

Tableau did not need 9,426 records to draw the view, and did not request them from the data source until the Underlying data tab was clicked.

Database engines are optimized to perform aggregations on data. Typically, these database engines are also located on powerful servers. Tableau leverages the optimization and power of the underlying data source. In this way, Tableau can visualize massive datasets with relatively little local processing of the data.

Additionally, Tableau will only query the data source when you make changes requiring a new query or a view refresh. Otherwise, it will use the aggregate results stored in a local cache, as illustrated here:


1st Rendering
Tableau
Cache
VizQL
SQL
or MDX
or TQL
Aggregated
Results
Data Source


Subsequent Renderings
Tableau
Cache
Data Source

In the preceding example, the query based on the fields in the view (that is, region as a dimension and the sum of sales as a measure) will only be issued once to the data source. When the four rows of aggregate results are returned, they are stored in the cache. Then, if you were to move Region to another visual encoding shelf, such as color, or Sales to a different visual encoding shelf, such as size, then Tableau will retrieve the aggregate rows from the cache and simply re-render the view.

*You can force Tableau to bypass the cache and refresh the data from a data source by pressing F5, or selecting your data source from the Data menu and selecting Refresh. Do this any time you want a view to reflect the most recent changes in a live data source.*

Of course, if you were to introduce new fields into the view that did not have cached results, then Tableau would send a new query to the data source, retrieve the aggregate results, and add those results to the cache.

## Connecting to data

There is virtually no limit to the data that Tableau can visualize! Almost every new version of Tableau adds new native connections. Tableau continues to add native connectors for cloud-based data. The **web data connector** allows you to write a connector for any online data you wish to retrieve. Additionally, for any database without a native connection, Tableau gives you the ability to use a generic ODBC connection. The **Extract API** allows you to programmatically extract and combine any data sources for use in Tableau.

You may have multiple data source connections to different sources in the same workbook. Each connection will show up under the Data tab on the left sidebar.

This section will focus on a few practical examples of connecting to various data sources. We won't cover every possible connection, but will cover several that are representative of others. You may or may not have access to some of the data sources in the following examples. *Don't worry if you aren't able to follow each example*. Merely observe the differences.

*Starting with Tableau 10, a* **connection** *technically refers to the connection made to a single set of data, such as tables in a single database, or files in a directory. A* **data source** *may contain more than one connection that can be joined together, such as a table in SQL Server joined to an Excel table. You may often hear these terms used interchangeably.*

**Connecting to data in a file**

File-based data includes all sources of data where the data is stored in a file. File-based data sources include the following:

**Extracts**: A .hyper or .tde file containing data that was extracted from an original source.

**Microsoft Access**: An .mdb or .accdb database file created in Access.

**Microsoft Excel:** An .xls, .xlsx, or .xlsm spreadsheet created in Excel. Multiple Excel sheets or sub-tables may be joined or unioned together in a single connection.

**Text file**: A delimited text file, most commonly .txt, .csv, or .tab. Multiple text files in a single directory may be joined or unioned together in a single connection.

**Local cube file**: A .cub file that contains multi-dimensional data.

These files are typically exported from OLAP databases.

**Adobe PDF**: A .pdf file that may contain tables of data that can be parsed by Tableau.

**Spatial file**: A .kml, .shp, .tab, .mif, or .geojson file that contains spatial objects that can be rendered by Tableau.

**Statistical file**: An .sav, .sas7bdat, .rda, or .rdata file generated by statistical tools, such as SAS or R.

**JSON file**: A .json file that contains data in JSON format.

In addition to those mentioned previously, you can connect to Tableau files to import connections that you have saved in another Tableau workbook (.twb or .twbx). The connection will be imported and changes will only affect the current workbook.

Follow this example to see a connection to an Excel file:

1. Navigate to the Connect to Excel sheet in the Chapter 02 Starter.twbx workbook.
2. From the menu, select Data | Create new data source and select Excel from the list of possible connections.
3. In the open dialogue, open the Superstore.xlsx file from the \Learning Tableau\Chapter 02 directory. Tableau will open the Data Source screen. You should see the two sheets of the Excel document listed on the left.
4. Double-click the Orders sheet and then the Returns sheet. Your data source screen should look similar to the following screenshot:

Take some time to familiarize yourself with the **Data Source** screen interface, which has the following features (numbered in the preceding screenshot):

1. Toolbar: The toolbar has a few of the familiar controls, including undo, redo, and save. It also includes the option to refresh the current data source.
2. Connections: All the connections in the current data source. Click Add to add a new connection to the current data source. This allows you to join data across different connection types. Each connection will be color-coded so that you can distinguish what data is coming from which connection.
3. Sheets (or Tables): This lists all the tables of data available for a given connection. This includes sheets, sub-tables, and named

ranges for Excel; tables, views, and stored procedures for relational databases; and other connection-dependent options, such as New Union or Custom SQL.

4. Data Source Name: This is the name of the currently selected data source. You may select a different data source using the drop-down arrow next to the database icon. You may click the name of the data source to edit it.
5. Connection Editor: Drop sheets and tables from the left into this area to make them part of the connection. For many connections, you may add multiple tables that will be joined or unioned together. We'll take a look at some advanced examples of options later in the chapter. For now, notice that you can hover over tables in this space and get options via a drop-down menu.
6. Live or Extract Options: For many data sources, you may choose whether you would like to have a live connection or an extracted connection. We'll look at these in further detail later in the chapter.
7. Data Source Filters: You may add filters to the data source. These will be applied at the data-source level, and thus to all views of the data using this data source in the workbook.
8. Preview Pane Options: These options allow you to specify whether you'd like to see a preview of the data or a list of metadata, and how you would like to preview the data (examples include alias values, hidden fields shown, and how many rows you'd like to preview).
9. Preview Pane/Metadata View: Depending on your selection in the options, this space either displays a preview of data or a list of all fields with additional metadata. Notice that these views give you a wide array of options, such as changing data types, hiding or renaming fields, and applying various data transformation functions. We'll consider some of these options in this and later

chapters.

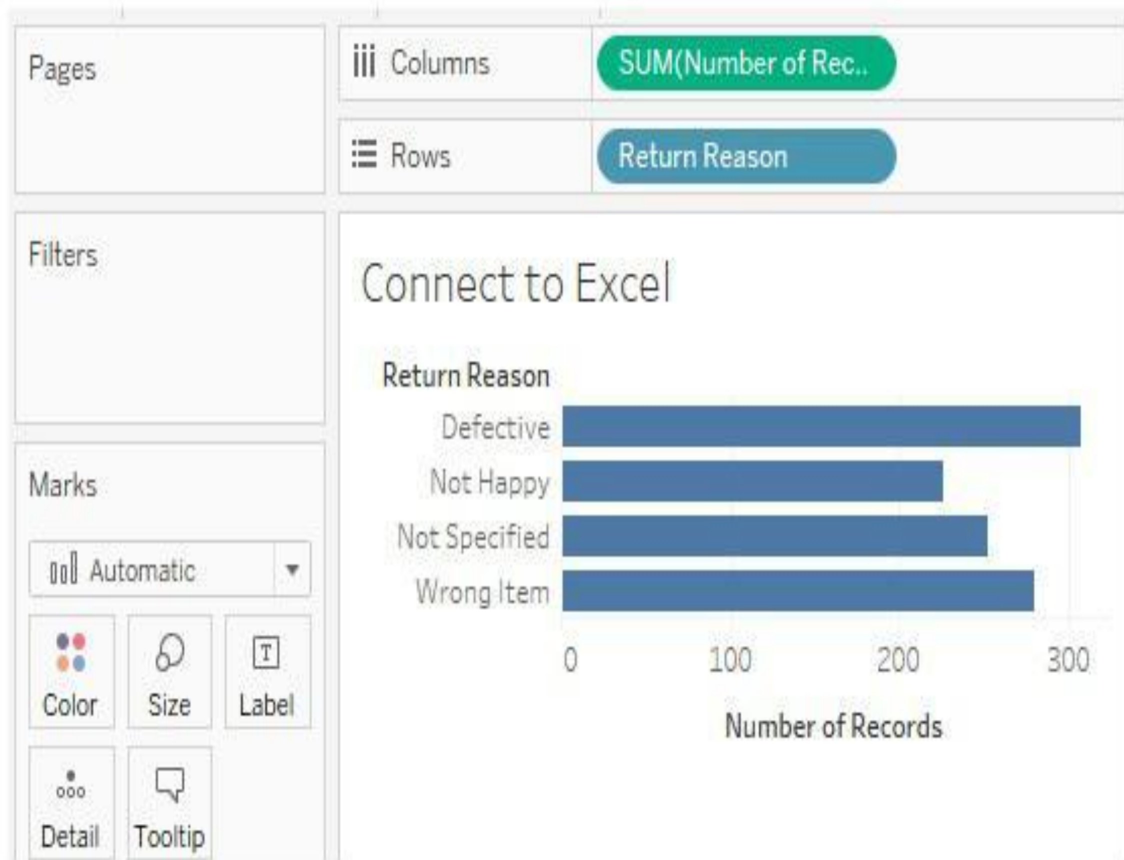
*Once you have created and configured your data source, you may click any sheet to start using it.*

Conclude this exercise with the following steps:

1. Click the data source name to edit the text and rename the data

   source to Orders and Returns.

2. Navigate to the Connect to Excel sheet and, using the Orders and Returns data source, create a time series showing **Number of Records** by **Return Reason**. Your view should look like the following screenshot:



3. As the connection you created is based on an inner join of Orders and Returns, this view shows the number of returns for each reason code.

If you need to edit the connection at any time, select Data from the menu, locate your connection, and then select Edit Data Source.... Alternately,

you may right-click any data source under the Data tab on the left sidebar and select Edit Data Source..., or click the Data Source tab in the lower-left. You may access the data source screen at any time by clicking the Data Source tab in the lower-left corner of Tableau Desktop.

**Connecting to data on a server**

Database servers, such as SQL Server, MySQL, Vertica, and Oracle, host data on one or more server machines and use powerful database engines to store, aggregate, sort, and serve data based on queries from client applications. Tableau can leverage the capabilities of these servers to retrieve data for visualization and analysis. Alternately, data can be extracted from these sources and stored in an extract (.hyper or .tde).

As an example of connecting to a server data source, we'll demonstrate connecting to SQL Server. If you have access to a server-based data source, you may wish to create a new data source and explore the details. However, there is no specific example to follow in the workbook in this chapter. As soon as the Microsoft SQL Server connection is selected, the interface displays options for some initial configuration as follows:

# Microsoft SQL Server

Server: TDS-W541-JM\AGAPE

Database: Optional

Enter information to sign in to the database:

- (•) Use Windows Authentication (preferred)
- ( ) Use a specific username and password:

Username:

Password:

- [ ] Require SSL
- [ ] Read uncommitted data

Initial SQL...

Sign In

A connection to SQL Server requires the Server name, as well as authentication information.

A database administrator can configure SQL Server to use Windows Authentication or a SQL Server username and password. With SQL Server, you can also optionally allow for reading uncommitted data. This can potentially improve performance, but may also lead to unpredictable results if data is being inserted, updated, or deleted at the same time Tableau is querying. Additionally, you may specify SQL to be run at

connect time using the Initial SQL... link in the lower-left corner.

*In order to maintain high standards of security, Tableau will not save a password as part of a data source connection. This means that if you share a workbook using a live connection with someone else, they will need to have credentials to access the data. This also means that when you first open the workbook, you will need to re-enter your password for any connections requiring a password.*

Once you click the orange Sign In button, you will see a screen that is very similar to the connection screen you saw for Excel. The main difference is on the left, where you have an option for selecting a database, as shown in the following screenshot:

**Connections** Add

TDS-W541-JM\TDS
Microsoft SQL Server

**Database**

Hospital

**Table**

- Discharge Details
- Hospital Visit
- Hospital_Visits
- Patient
- Patient Visit
- Primary Physician
- v_Patients
- v_Visits

- New Custom SQL
- New Union

**Stored Procedures**

- Get_Active_Doctors
- Get_Patient_Diagnoses
- Get_Patient_Treatments

Once you've selected a database, you will see the following:

Table: This shows any data tables or views contained in the selected database.

New Custom SQL: You may write your own custom SQL scripts and add them as tables. You may join these as you would any other table or view.

New Union: You may union together tables in the database. Tableau will match fields based on name and data type, and you may additionally merge fields as needed.

Stored Procedures: You may use a stored procedure that returns a table of data. You will be given the option of setting values for stored procedure parameters, or using or creating a Tableau parameter to pass values.

Once you have finished configuring the connection, click a tab for any sheet to begin visualizing the data.

### Connecting to data in the cloud

Certain data connections are made to data that is hosted in the cloud. These include **Amazon Redshift**, **Google Analytics**, **Google Sheets**, **Salesforce**, and many others. It is beyond the scope of this book to cover each connection in depth, but as an example of a cloud data source, we'll consider connecting to Google Sheets.

Google Sheets allows users to create and maintain spreadsheets of data online. Sheets may be shared and collaborated on by many different users. Here, we'll walk through an example of connecting to a sheet that is shared via link.

To follow the example, you'll need a free Google account. With your credentials, follow these steps:

1. Click the Add new data source button on the toolbar, as shown here:

2. Select Google Sheets from the list of possible data sources. You may use the search box to quickly narrow the list.
3. On the next screens, sign into your Google Account and allow Tableau Desktop the appropriate permissions. You will then be presented with a list of all your Google Sheets, along with preview and search capabilities, as shown in the following screenshot:

Select Your Google Sheet

Signed in as milligan Sign Out

Search by sheet name or enter URL | Search

| Name | Owned by | Last Opened By Me |
|---|---|---|
| Greek New Testament | Joshua Milligan | Mar 12, 2018 |
| Polygonic Hex Map.xlsx | Joshua Milligan | Dec 20, 2017 |
| SMS received | Joshua Milligan | Sep 25, 2017 |
| Teknion Sign Up Sheet August – December 2017 | | Aug 11, 2017 |
| Superstore | Joshua Milligan | Jul 10, 2017 |
| Company Profit | Joshua Milligan | Jun 27, 2017 |
| Test_1 | Joshua Milligan | Dec 5, 2017 |
| SMS received | Joshua Milligan | Sep 25, 2017 |
| Assimilations | Borg Queen | Aug 11, 2017 |
| Superstore | Joshua Milligan | Jul 10, 2017 |
| Company Profit | Joshua Milligan | Jun 27, 2017 |

Greek New Testament

Last Modified On **Dec 7, 2017**

Last Modified By **Joshua Milligan**

Open in Google Drive

Cancel | Connect

4. Enter this URL (for convenience, it is included in the Chapter 02

Starter workbook in the Connect to Google Sheets tab, and may be copied and pasted) into the search box and click the **Search**

button: https://docs.google.com/spreadsheets/d/1fWMGkPt0o7sdbW50tG4QLSZDwkjNO9X0mCkw-LKYu1A/edit?usp=sharing.

5. Select the resulting Superstore sheet in the list and then click the Connect button. You should now see the **Data Source** screen.
6. Rename the data source to Superstore (Google Sheets).
7. For the purpose of this example, switch the connection option from Live to Extract. When connecting to your own Google Sheets data, you may choose either Live or Extract.
8. Navigate to the Connect to Google Sheets sheet. The data should be extracted within a few seconds.
9. Create a filled map of **Profit** by **State**, with **Profit** defining the **Color** and the **Label**.

**Shortcuts for connecting to data**

You can make certain connections very quickly. These options will allow you to begin analyzing more quickly:

- **Paste data from the clipboard**. If you have copied data from a spreadsheet, a table on a webpage, or a text file, you can often paste the data directly into Tableau. This can be done using *Ctrl + V*, or Data | Paste Data from the menu. The data will be stored as a file and you will be alerted to its location when you save the workbook.

- **Select File | Open from the menu**. This will allow you to open any data file that Tableau supports, such as text files, Excel files, Access files (not available on macOS), spatial files, statistical files, JSON, and even offline cube (.cub) files.

- **Drag and drop a file from Windows Explorer or Finder onto the Tableau workspace**. Any valid file-based data source can be dropped onto the Tableau workspace, or even the Tableau shortcut on your desktop or taskbar.

- **Duplicate an existing connection**. You can duplicate an existing data source connection by right-clicking and selecting Duplicate.

**Managing data source metadata**

Data sources in Tableau store information about the connection(s). In addition to the connection itself (example, database server name, database, and/or file names), the data source also contains information about all the fields available (such as field name, data type, default format, comments, and aliases). Often, this *data about the data* is referred to as **metadata**.

Right-clicking a field in the data pane reveals a menu of metadata options. Some of these options will be demonstrated in a later exercise; others will be explained throughout the book. These are some of the options available via right-clicking:

- Renaming the field
- Hiding the field
- Changing aliases for values of dimension (other than date fields)
- Creating calculated fields, groups, sets, bins, or parameters
- Splitting the field
- Changing the default use of a date or numeric field to either discrete or continuous

Redefining the field as a dimension or a measure

- Changing the data type of the field
- Assigning a geographic role of the field

- Changing defaults for how a field is displayed in a visualization, such as the default colors and shapes, number or date format, sort order (for dimensions), or type of aggregation (for measures)

- Adding a default comment for a field (which will be shown as a tooltip when hovering over a field in the data pane, or shown as part of the description when Describe... is selected from the menu)

- Adding or removing the field from a hierarchy

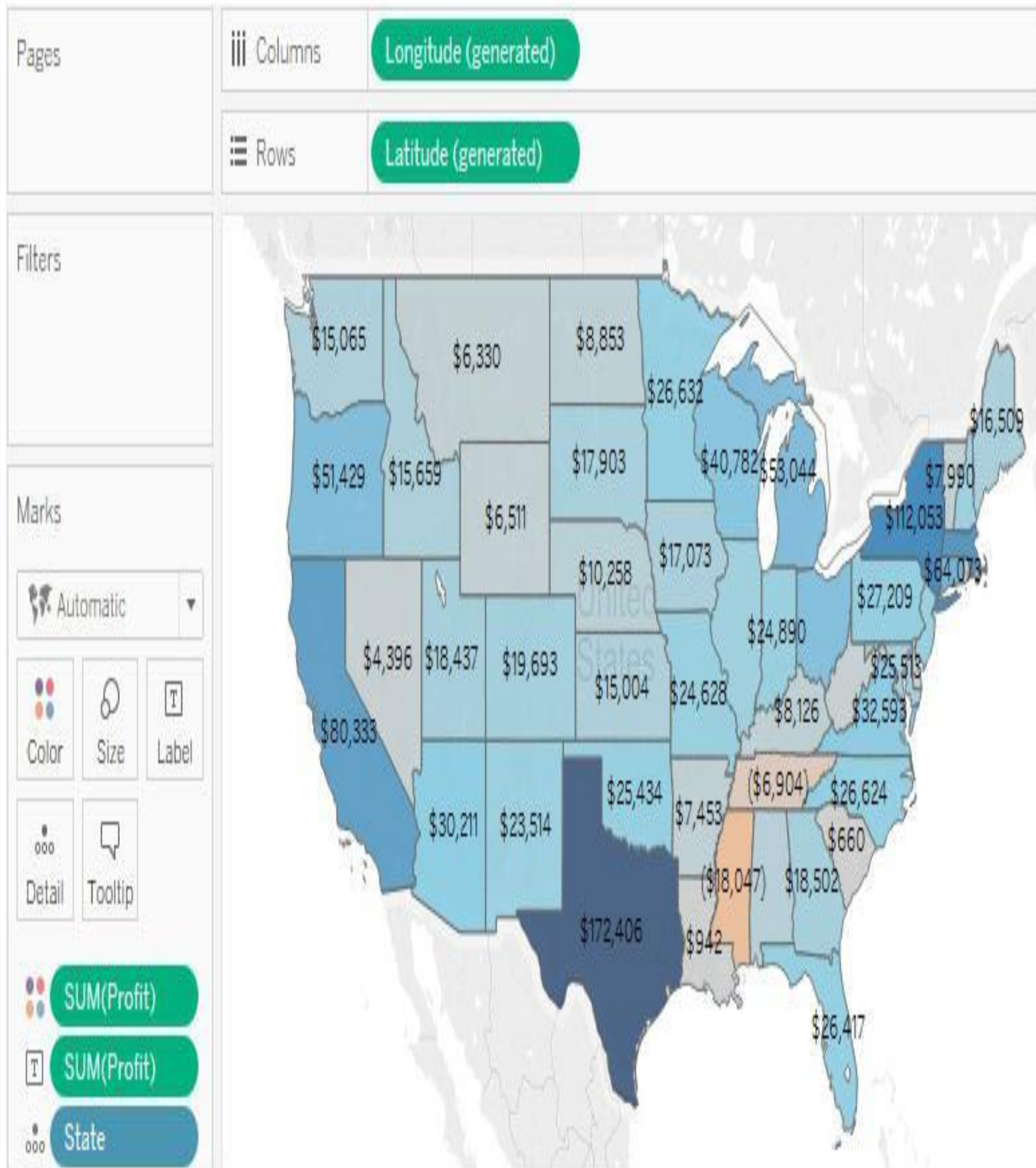
*Metadata options that relate to the visual display of the field, such as default sort order or default number format, define the overall default for a field. However, you can override the defaults in any individual view by right-clicking the active field on the shelf and selecting the desired options.*

To see how this works, use the filled map view of **Profit** by **State** that you created in the Connect to Google Sheets view. If you did not create this view, you may use the Orders and Returns data source, though the resulting view will be slightly different. With the filled map in front of you, follow these steps:

1. Right-click the Profit field in the data pane and select Default Properties | Number Format.... The resulting dialog gives you many options for numeric format.
2. Set the number format to Currency (Custom) with 0 Decimal places. After clicking OK, you should notice that the labels on the map have updated to include currency notation.
3. Right-click the Profit field again and select Default properties | Color.... The resulting dialog gives you an option to select and customize the default color encoding of the Profit field.

   Experiment with various palettes and settings. Notice that every time you click the Apply button, the visualization updates.

*Diverging palettes (palettes that blend from one color to another) work particularly well for fields such as Profit, which can have negative and positive values. The default center of 0 allows you to fairly easily tell what values are positive or negative based on the color shown.*



Because you have set the default format for the field at the data-source level, any additional views you create using Profit will include the default formatting you specified.

*Consider using color blind-safe colors in your visualizations. Orange and blue are*

*usually considered one color blind-safe alternative to red and green. Tableau also includes a discrete color-blind safe palette. Additionally, consider adjusting the intensity of the colors.*

**Working with extracts instead of live connections**

Most data sources allow the option of either connecting live or extracting the data. However, some cloud-based data sources require an extract. Conversely, OLAP data sources cannot be extracted and require live connections.

When using a live connection, Tableau issues queries directly to the data source (or uses data in the cache, if possible). When you extract the data, Tableau pulls some or all of the data from the original source and stores it in an extract file. Prior to version 10.5, Tableau used a Tableau Data Extract (.tde) file. Starting with version 10.5, Tableau uses Hyper extracts (.hyper) and will convert .tde files to .hyper as you update older workbooks.

Extracts extend the way in which Tableau works with data. Consider the following diagram:

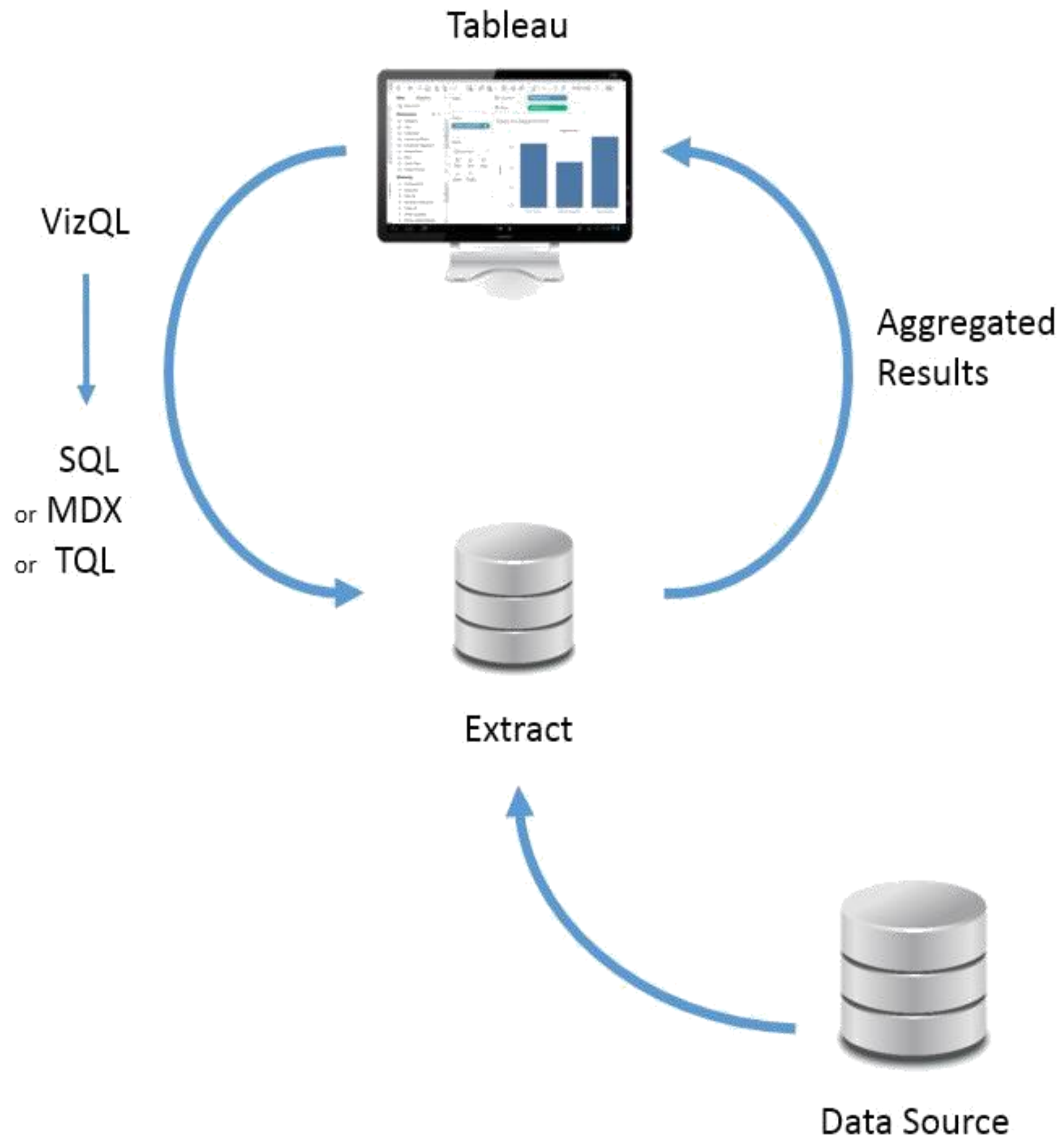


Tableau
VizQL
SQL
or MDX
or TQL
Aggregated
Results
Extract
Data Source

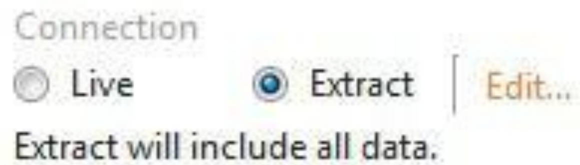
The fundamental paradigm of how Tableau works with data does not change, but you'll notice that Tableau is now querying and getting results from the extract. Data can be retrieved from the source again to refresh the extract. Thus, each extract is a snapshot of the data source at the time of the latest refresh. Extracts offer the benefit of being portable and extremely efficient.

**Creating extracts**

Extracts can be created in multiple ways, as follows:

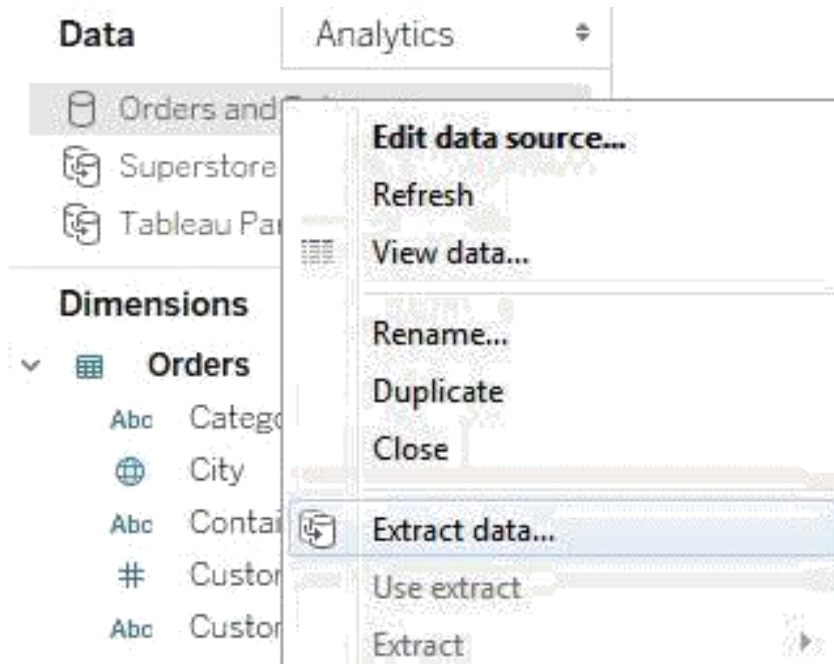
- Select Extract on the Data Source screen as follows. The Edit...

link will allow you to configure the extract:




- Select the data source from the Data menu, or right-click the data source on the data pane and select Extract data.... You will be

given a chance to set configuration options for the extract, as demonstrated in the following screenshot:




- Developers may create an extract using the Tableau Data Extract API. This API allows you to use Python or C/C++ to programmatically create an extract file. The details of this approach are beyond the scope of this book, but documentation is readily available on Tableau's website.
- Certain tools, such as Alteryx or Tableau Prep, are able to output Tableau extracts.

When you first create or subsequently configure an extract, you will be prompted to select certain options, as shown here:

Extract Data

Specify how much data to extract:

Filters (Optional)

| Filter | Details |
|---|---|
| Region | keeps Central and South |
| Category | keeps Office Machines |

Add... Edit... Remove

Aggregation

☐ Aggregate data for visible dimensions

☐ Roll up dates to Year

Number of Rows

◉ All rows

☑ Incremental refresh

Identify new rows using column: Row ID

All rows will be added. Full Refresh

○ Top: rows

History... Hide All Unused Fields Extract Cancel

You have a great deal of control when configuring an extract. Here are the various options, and the impact your choices will make on performance and flexibility:

You may optionally add **Extract** filters, which limit the extract to a subset of the original source. In this example only, records where Region is **Central** or **South** and where Category is **Office Machines**

will be included in the extract.

You may aggregate an extract by checking the box. This means that data will be rolled up to the level of visible dimensions and, optionally, to a specified date level, such as year or month.

**Visible fields** *are those that are shown in the data pane. You may hide a field from the* **Data Source** *screen or from the data pane by right-clicking a field and selecting Hide. This option will be disabled if the field is used in any view in the workbook. Hidden fields are not available to be used in a view. Hidden fields are not included in an extract as long as they are hidden prior to creating or optimizing the extract.*

In the preceding example, if only the Region and Category dimensions were visible, the resulting extract would only contain two rows of data (one row for **Central** and another for **South**). Additionally, any measures would be aggregated at the Region/Category level and would be done with respect to the **Extract** filters. For example, Sales would be rolled up to the sum of sales in **Central/Office Machines** and **South/Office Machines**. All measures are aggregated according to their default aggregation.

You may adjust the Number of Rows in the extract by including all rows or a sampling of the top **N** rows in the dataset. If you select all rows, you can indicate an incremental refresh. If your source data incrementally adds records, and you have a field such as an identity column or date field that can be used reliably to identify new records as they are added, then an incremental extract can allow you to add those records to the extract without recreating the entire extract. In the preceding example, any new rows where **Row ID** is higher than the highest value of the previous extract refresh would be included in the next incremental

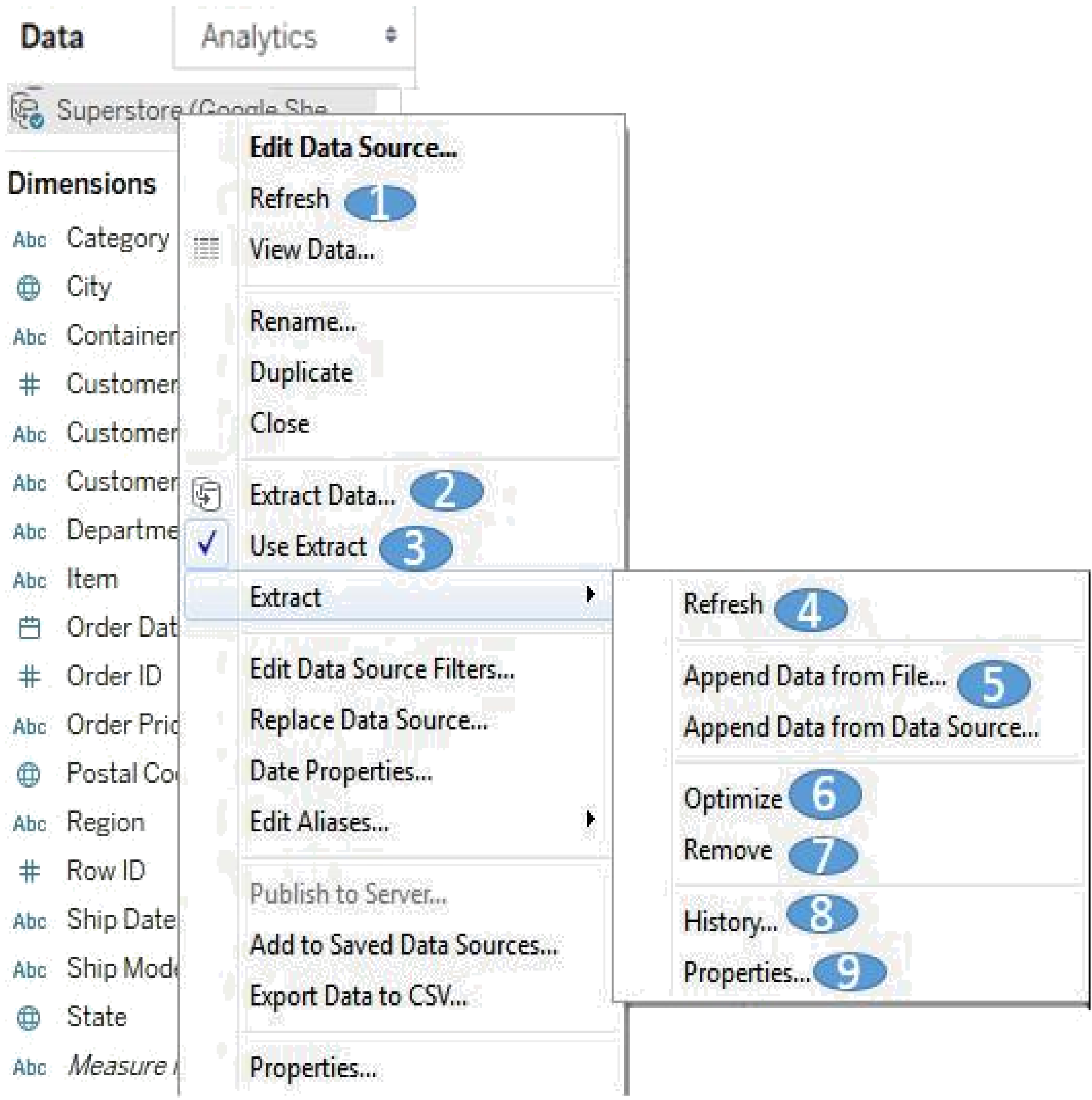
refresh.

*Incremental refreshes can be a great way to deal with large volumes of data that grow over time. However, use incremental refreshes with care, because the incremental refresh will only add new rows of data based on the field you specify. You won't get changes to existing rows, nor will rows be removed if they were deleted at the source. You will also miss any new rows if the value for the incremental field is less than the maximum value in the existing extract.*

**Using extracts**

Any data source that is using an extract will have a distinctive icon that indicates the data has been pulled from an original source into an extract, as shown in the following screenshot:

The first data connection in the preceding data pane is extracted, while the second is not. After an extract has been created, you may choose to use the extract or not. When you right-click a data source (or Data from the menu and then the data source), you will see the following menu options, as demonstrated in this screenshot:


Data
Analytics
Superstore (Google She
Dimensions
Category
City
Container
Customer
Customer
Customer
Departme
Item
Order Dat
Order ID
Order Pric
Postal Co
Region
Row ID
Ship Date
Ship Mod
State
Measure
Edit Data Source...
Refresh 1
View Data...
Rename...
Duplicate
Close
Extract Data... 2
Use Extract 3
Extract
Edit Data Source Filters...
Replace Data Source...
Date Properties...
Edit Aliases...
Publish to Server...
Add to Saved Data Sources...
Export Data to CSV...
Properties...
Refresh 4
Append Data from File... 5
Append Data from Data Source...
Optimize 6
Remove 7
History... 8
Properties... 9

1. Refresh: The Refresh option under the data source simply tells Tableau to refresh the local cache of data. With a live data source, this would requery the underlying data. With an extracted source, the cache is cleared and the extract is requeried, but this Refresh option does not update the extract from the original source. To do that, use Refresh under the Extract sub-menu (see step 4 in this list).
2. Extract data...: This creates a new extract from the data source (replacing an existing extract if it exists).

3. Use Extract: This option is enabled if there is an extract for a given data source. Unchecking the option will tell Tableau to use a live connection instead of the extract. The extract will not be removed and may be used again by checking this option at any time. If the original data source is not available to this workbook, then Tableau will ask where to find it.

4. Refresh: This Refresh option refreshes the extract with data from the original source. It does not optimize the extract for some changes you make (such as hiding fields or creating new calculations).
5. Append data from file...: This option allows you to append additional files to an existing extract, provided they have the same exact data structure as the original source. This adds rows to your existing extract; it will not add new columns.
6. Optimize: This will restructure the extract, based on changes you've made since originally creating the extract, to make it as efficient as possible. For example, certain calculated fields may be **materialized** (that is, calculated once so that the resulting value can be stored) and newly hidden columns or deleted calculations will be removed from the extract.
7. Remove: This removes the definition of the extract, optionally deletes the extract file, and resumes a live connection to the original data source.
8. History: This allows you to view the history of the extract and refreshes.
9. Properties: This enables you to view the properties of the extract, such as the location, underlying source, filters, and row limits.

**Performance**

Prior to 10.5, Tableau Data Extracts (.tde files) were very efficient columnar databases that performed well with the Tableau data engine. With Tableau 10.5, Tableau introduced the Hyper data engine, which shows remarkable performance gains, especially for large datasets. Both

.tde and .hyper extracts will perform faster than most traditional live database connections. This is based on several factors, as follows:

Hyper extracts make use of a hybrid of OLTP and OLAP models, and the engine determines the optimal query. Tableau Data Extracts are columnar and also very efficient to query.

Extracts are structured so they can be loaded quickly into memory without additional processing and moved between memory and disk storage, so the size is not limited to the amount of RAM available.●

Many calculated fields are materialized in the extract. The pre-calculated value stored in the extract can often be read faster than executing the calculation every time the query is executed. Hyper extracts extend this by potentially materializing many aggregations.

You may choose to use extracts to increase performance over traditional databases. To maximize your performance gain, consider the following actions:

Prior to creating the extract, hide unused fields. If you have

created all desired visualizations, you can click the Hide Unused Fields button on the extract dialog to hide all fields not used in any view or calculation.

If possible, use a subset of data from the original source. For example, if you have historical data for the last 10 years, but will only need the last two years for analysis, then filter the extract by the Date field.

Optimize an extract after creating or editing calculated fields, or deleting or hiding fields.

Store extracts on solid state drives.

**Portability and security**

Let's say that your data is hosted on a database server accessible only from inside your office network. Normally, you'd have to be onsite or using a VPN to work with the data. With an extract, you can take the data with you and work offline.

An extract file contains data extracted from the source. When you save a workbook, you may save it as a Tableau workbook (.twb) file or a Tableau Packaged Workbook (.twbx) file. A workbook (.twb) contains definitions for all the connections, fields, visualizations, and dashboards, but does not contain any data or external files, such as images. When you save a packaged workbook (.twbx), any extracts and external files are packaged together in a single file with the workbook.

A packaged workbook using extracts can be opened with Tableau Desktop, Tableau Reader, and published to Tableau Public or Tableau Online.

*A packaged workbook file (.twbx) is really just a compressed .zip file. If you rename the extension from .twbx to .zip, you can access the contents as you would any other*

*.zip file. You may also consider associating the .twbx extension with your ZIP utility so you won't have to rename the files.*

There are a couple of security considerations to keep in mind when using an extract:

- 

The extract is made using a single set of credentials. Any security layers that limit which data can be accessed according to the credentials used will not be effective after the extract is created. An extract does not require a username or password. All data in an extract can be read by anyone.

Any data for visible (non-hidden) fields contained in an extract

file (.hyper or .tde), or an extract contained in a packaged workbook (.twbx), can be accessed even if the data is not shown in the visualization. Be very careful to limit access to extracts or packaged workbooks containing sensitive or proprietary data.

The story is told of an employee who sent a packaged workbook containing HR data to others in the company. Even though none of the dashboards displayed sensitive data, the extract contained it. It wasn't long before everyone in the company knew everyone else's salary and the original individual was no longer an employee.

**When to use an extract**

You should consider various factors when determining whether or not to use an extract. In some cases, you won't have an option (for example, OLAP requires a live connection and some cloud-based data sources require an extract). In other cases, you'll want to evaluate your options.

In general, use an extract when:

You need better performance than you can get with the live connection.

You need the data to be portable.

You need to use functions that are not supported by the database data engine (for example, MEDIAN is not supported with a live connection to SQL Server).

You want to share a packaged workbook. This is especially true if you want to share a packaged workbook with someone who uses the free Tableau Reader, which can only read packaged workbooks with data extracted.

In general, do not use an extract when you have any of the following use cases:

You have sensitive data that should not be accessible by certain users, or you have no control over who will be able to access the extract. However, you may hide sensitive fields prior to creating the extract, in which case they are no longer part of the extract.

- You need to manage security based on login credentials. (However, if you are using Tableau Server, you may still use extracted connections hosted on Tableau Server that are secured by login. We'll consider sharing your work with Tableau Server in Chapter 12, *Sharing Your Data Story*).
- You need to see changes in the source data updated in real time.
- The volume of data makes the time required to build the extract impractical. The number of records that can be extracted in a reasonable amount of time will depend on factors such as the data types of fields, the number of fields, the speed of the data source, and network bandwidth. The hyper engine typically builds the new .hyper extracts much faster than the older .tde files were built.

**Tableau file types**

In addition to the file types mentioned previously, there are quite a few other file types associated with Tableau. The following are some of the Tableau file types:

- .tbm: A Tableau Bookmark file—an XML file containing a definition of a static snapshot of a single view and associated data sources. As sheets can now be copied and pasted from one workbook to another, this file type is largely not needed. You can create bookmarks and import them into other workbooks from the Window menu.

- .hyper: A Hyper extract—a binary file containing data extracted from another source. This is the extract format used by Tableau 10.5 and later, which utilizes the much faster and scalable hyper engine.

- .tde: A Tableau Data Extract—a binary file containing data extracted from another source. The .tde file by itself does not contain information about the original data source. Tableau Data Extracts are used by versions prior to 10.5. Tableau 10.5 and later can read .tde files but will, under most circumstances, update them to the new .hyper format.

- .tds: Tableau Data Source file—an XML file containing the definition of a data source (the server name, file path, and so on), but does not contain the data. You can export a data source as a .tds file by right-clicking the data source and selecting Add to

Saved Data Sources. Any .tds files in your My Tableau Repository Data Sources directory will show as data connection shortcuts on the Home Screen.

- .tfl: A Tableau Flow file—a file defining a Tableau Prep flow.

Tableau Desktop does not read this file type.

- .tflx: A Packaged Tableau Flow file—a compressed .zip file containing the .tfl file and extracts of the file-based data sources for the flow. Tableau Desktop does not read this file type.

- .tdsx: A Tableau Data Source extracted file—a compressed file containing the definition of the data source, along with an extract of the data. You may create packaged data sources in the same way you create .tds files, selecting .tdsx as the file type.

- .tld/.tlf/.tlq/.tlr: A Tableau License file

—Disconnected/File/Request/Return/Response file types are used in

license activation.

- .tms: A Tableau Map Source file—an XML file type used to specify map services and configuration available to Tableau.

- .tmsd: A Tableau Map Source Defaults file—an XML file containing defaults for map services

- .tps: Tableau Preferences—an XML file containing preference defaults for Tableau Desktop, including UI elements and color palettes.

- .tsvc: The Tableau Atom Service file type.

- .twb: A Tableau Workbook—an XML file containing definitions for all sheets, data sources, preferences, and formatting. It does

not contain any data.

.twbx: A Tableau Packaged Workbook—a compressed file containing the Tableau workbook (.twb) file, along with any data extracts (.tde) and any other external files (such as images, or text/Excel files for data sources that are not extracted).

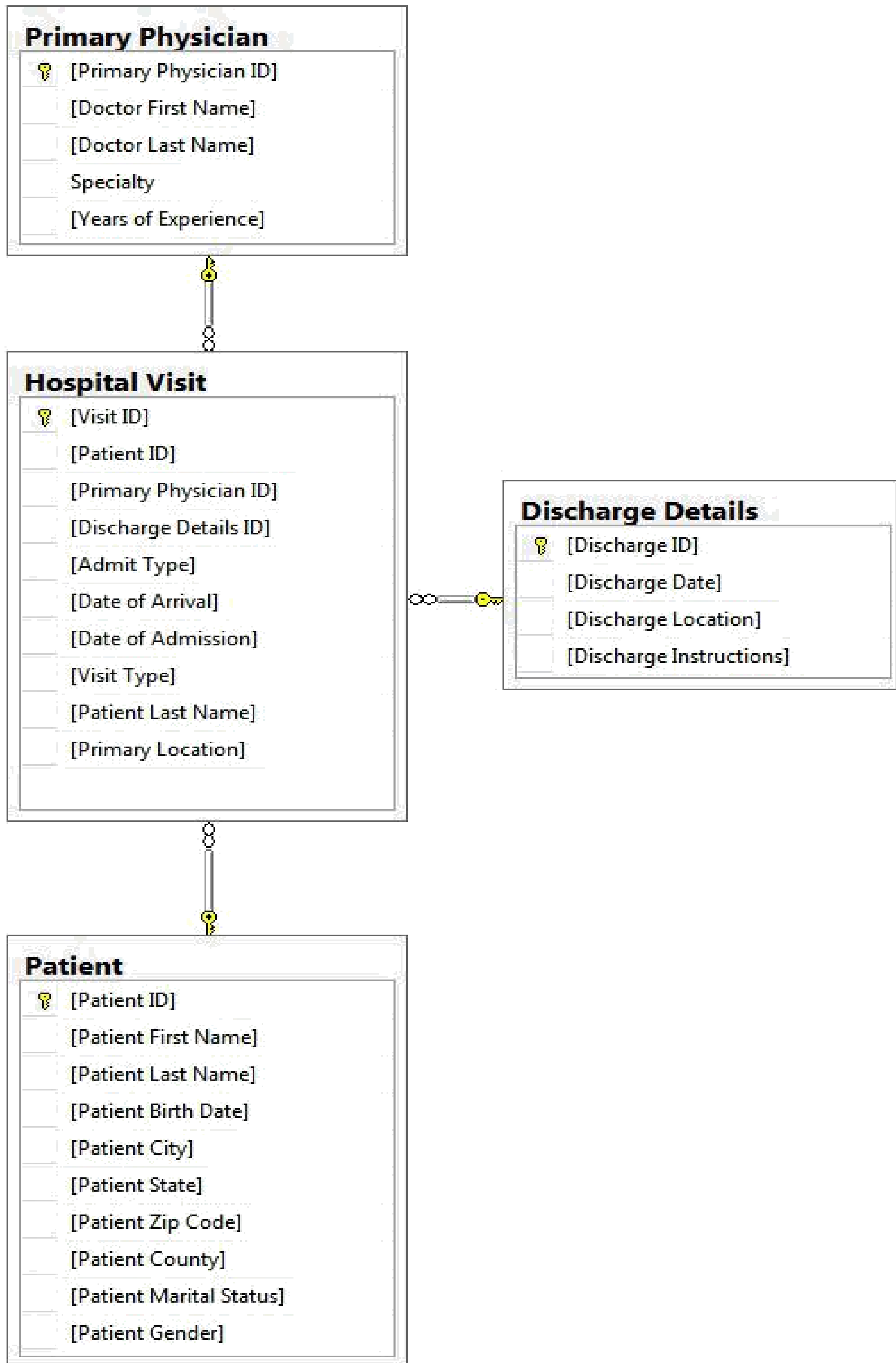
**Joins and blends**

Joining tables and blending data sources are two different ways to link related data together in Tableau. **Joins** are performed to link tables of data together on a row-by-row basis. **Blends** are performed to link together multiple data sources at an aggregate level.

**Joining tables**

Most databases have multiple tables of data that are related in some way. Additionally, with Tableau 10 and later, you are able to join together tables of data across various data connections for many different data sources. As we'll see, Tableau makes it very easy to join together tables of data relatively easy.

Consider, for example, tables such as these:

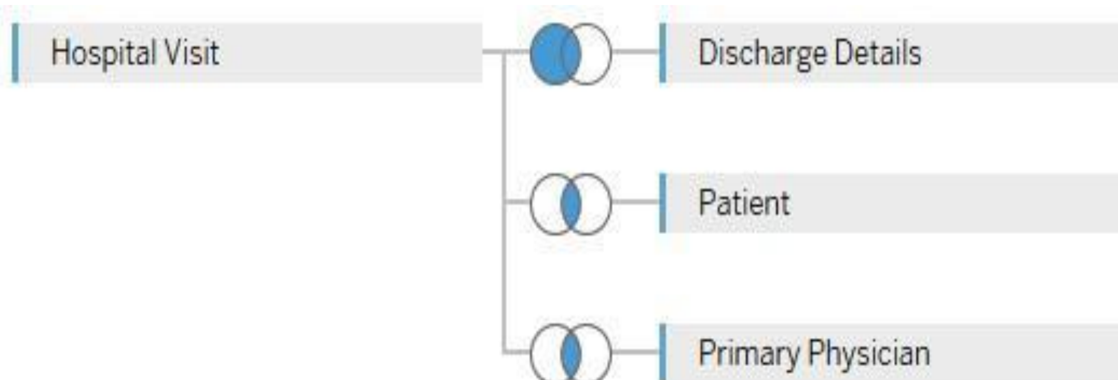

Primary Physician
[Primary Physician ID]
[Doctor First Name]
[Doctor Last Name]
Specialty
[Years of Experience]
Hospital Visit
[Visit ID]
[Patient ID]
[Primary Physician ID]
[Discharge Details ID]
[Admit Type]
[Date of Arrival]
[Date of Admission]
[Visit Type]
[Patient Last Name]
[Primary Location]
Discharge Details
[Discharge ID]
[Discharge Date]
[Discharge Location]
[Discharge Instructions]
Patient
[Patient ID]
[Patient First Name]
[Patient Last Name]
[Patient Birth Date]
[Patient City]
[Patient State]
[Patient Zip Code]
[Patient County]
[Patient Marital Status]
[Patient Gender]

The primary table is the **Hospital Visit** table, which has a record for every visit of a patient to the hospital and includes details such as admission type (examples include inpatient, outpatient, and ER). It also contains key fields that link a visit to a **Primary Physician**, **Patient**, and **Discharge Details**.

When you connect to the database in Tableau, you'll see the tables listed on the left and will have the option to drag and drop them into the data source designer.

*Typically, you'll want to start by dragging the primary table into the designer. In this case,* **Hospital Visit** *contains keys for joining additional tables. Those tables should be dragged and dropped after the primary table.*

If key fields and relationships have been defined in the database, Tableau will automatically create the joins as you add additional tables. Otherwise, it will attempt to match field names. In any case, you may adjust the joins as needed. The preceding tables will look similar to the following diagram when dropped into the designer:



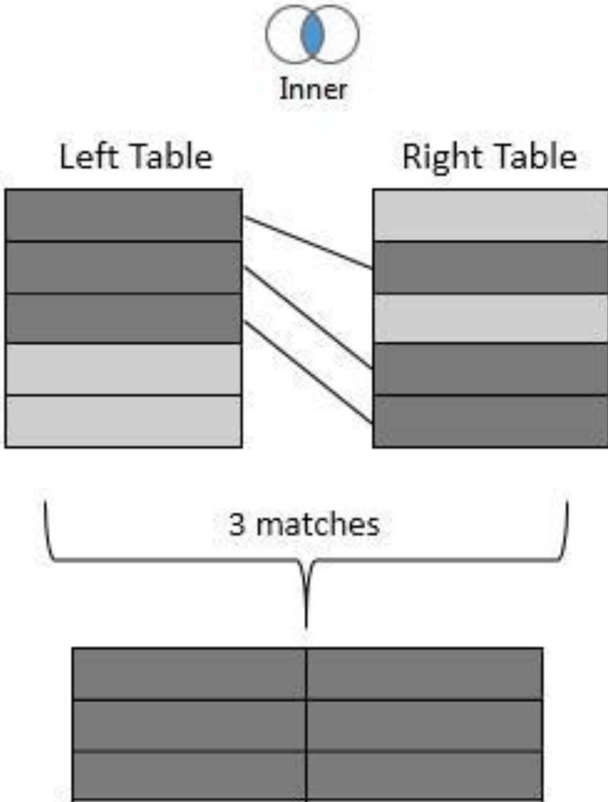
You may adjust the join by clicking the small diagram between the tables. The diagram indicates what kind of join is used. For example, the join between **Hospital Visit** and **Patient** is an **Inner Join** because it is assumed that every visit will have a patient and every patient will have a visit. However, the join between **Hospital Visit** and **Discharge Details** is a left join because some records in **Hospital Visit** may be for patients still in the hospital (so they haven't been discharged).

Clicking on the diagram will allow you to select a different type of join and define which fields are part of the join.

You may specify the following types of joins:

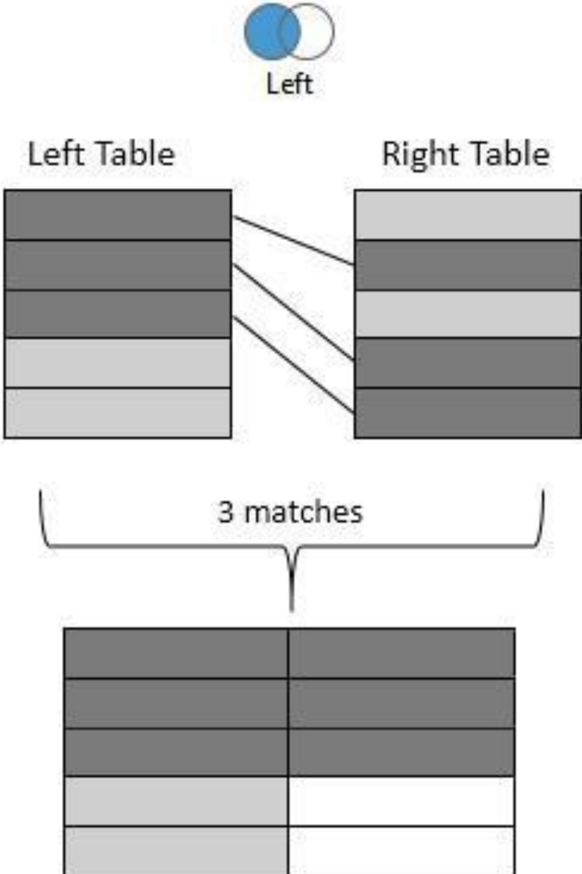
- 

**Inner**: Only records that match the join condition from both the table on the left and the table on the right will be kept. In the following example, only the three matching rows are kept in the results:

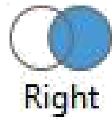


**Left**: All records from the table on the left will be kept. Matching records from the table on the right will have values in the resulting table, while unmatched records will contain NULL values for all fields from the table on the right. In the following example, the

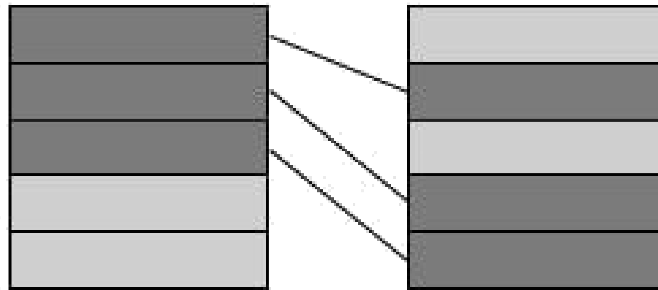
five rows from the left table are kept with NULL results for right values that were not matched:



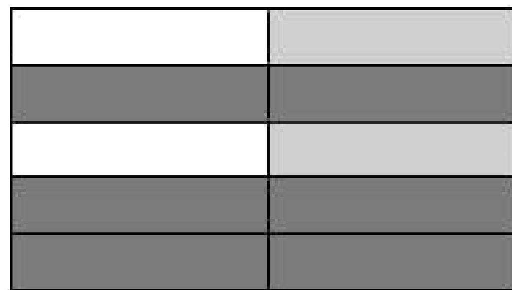
**Right**: All records from the table on the right will kept. Matching records from the table on the left will result in values, while unmatched records will contain NULL values for all fields from the table on the left. Not every data source supports a right join. If it is not supported, the option will be disabled. In the following example, the five rows from the right table are kept with NULL results for left values that were not matched:



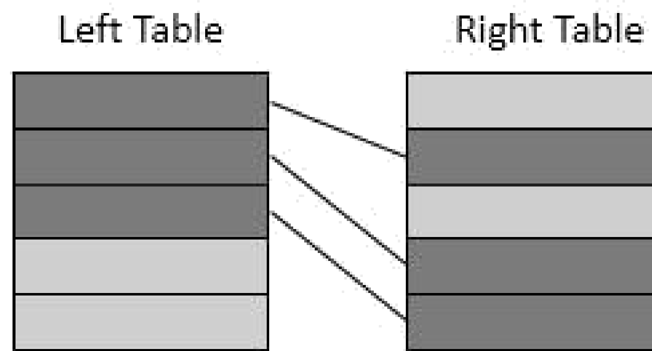

Left Table Right Table



3 matches



**Full Outer**: All records from tables on both sides will be kept. Matching records will have values from the left and the right. Unmatched records will have NULL values where either the left or the right matching record was not found. Not every data source supports a full outer join. If it is not supported, the option will be disabled. In the following example, all rows are kept from both sides with NULL values where matches were not found:

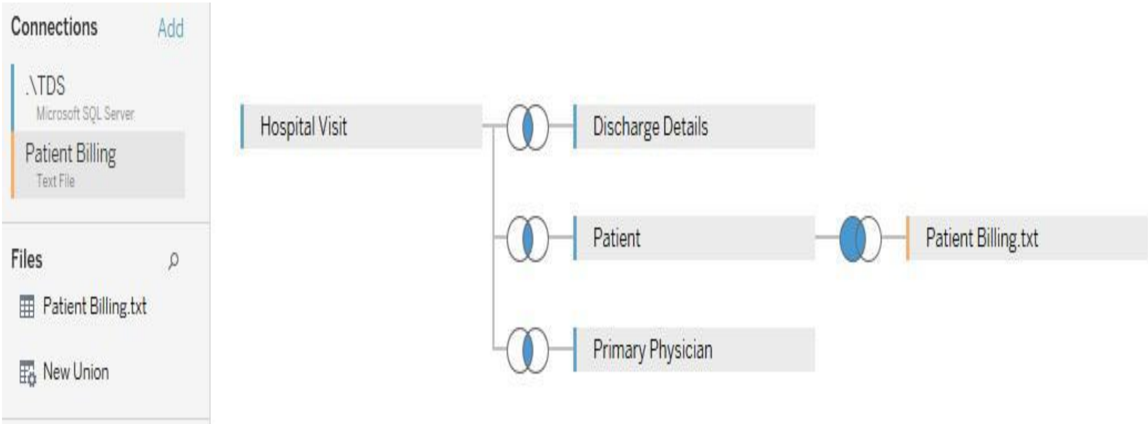



3 matches

**Cross database joins**

With Tableau, you have the ability to join (at a row level) across multiple different data connections. Joining across different data connections is referred to as a **cross database join**. For example, you can join SQL Server tables with text files or Excel files, or tables in one database with tables in another, even if those are on a different server. This opens up all kinds of possibilities for supplementing your data or analyzing data from disparate sources.

Consider the hospital data mentioned previously. It would not be uncommon for billing data to be in a separate system from patient care data. Let's say you had a file for patient billing that contained data you wanted to include in your analysis of hospital visits. You would be able to accomplish this by adding the text file as a data connection and then joining it to the existing tables, as follows:



You'll notice that the interface on the Data Source screen includes an Add link that allows you to add data connections to a data source. Clicking on each connection will allow you to drag and drop tables from that connection into the Data Source designer and specify the joins as you desire. Each data connection will be color-coded so that you can immediately identify the source of various tables in the designer.

In the preceding example, the Patient Billing.txt text file has been joined to the Patient table from SQL Server.

*With all joins, including cross-data connection joins, you will need to make certain that you have field(s) that are shared in common between the tables. For example, to join Patient Billing.txt to the Patient table, there will need to be some kind of patient ID or account number that can be matched. Cross database joins also require that the data types be identical. You can use a* **join calculation** *to change the type if needed.*

## Blending data sources

Data blending is a powerful and innovative feature in Tableau. It allows you to use data from multiple data sources in the same view. Often these sources may be different types. For example, you can blend data from Oracle with data from Excel. You can blend Google Analytics data with a spatial file. Data blending also allows you to compare data at different levels of detail. Some advanced uses of data blending will be covered in Ch apter 8, *Digging Deeper: Trends, Clustering, Distributions and Forecasting*. For now, let's consider the basics and a simple example.

Data blending is done at an aggregate level and involves different queries sent to each data source, unlike joining, which is done at a row level and involves a single query to a single data source. A simple data blending process involves several steps, as shown in the following diagram:

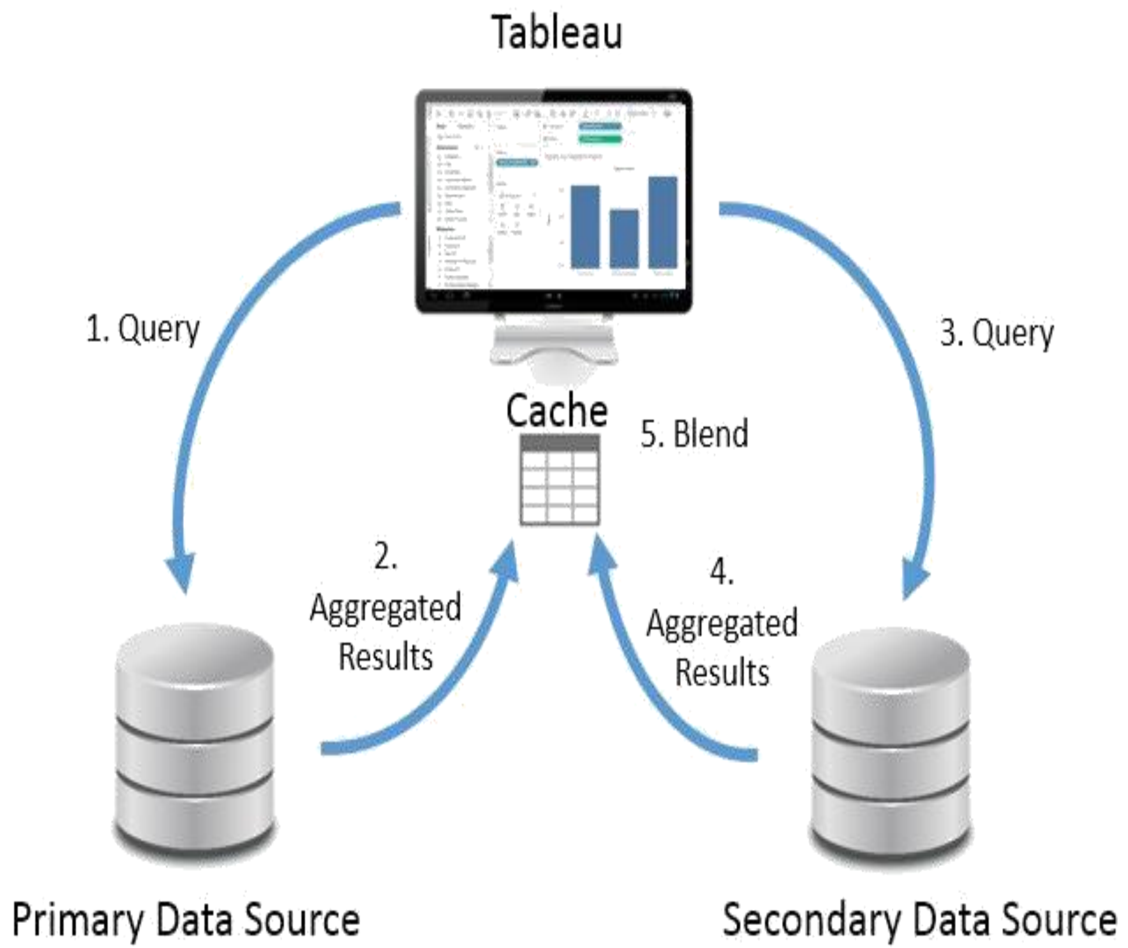


Tableau
1. Query
3. Query
Cache
5. Blend
2.
Aggregated
Results
4.
Aggregated
Results
Primary Data Source
Secondary Data Source

We can see the following from the preceding diagram:

- Tableau issues a query to the primary data source.
- The underlying data engine returns aggregate results.
- Tableau issues another query to the secondary data source. This query is filtered based on the set of values returned from the primary data source for dimensions that link the two data sources.
- The underlying data engine returns aggregate results from the secondary data source.
- The aggregated results from the primary data source and the

aggregated results from the secondary data source are blended together in the cache.

It is important to note how data blending is different from joining. Joins are accomplished in a single query and results are matched row-by-row. Data blending occurs by issuing two separate queries and then blending together the aggregate results.

There can only be one primary source, but there can be as many secondary sources as you desire. Steps three and four will be repeated for each secondary source. When all aggregate results have been returned, Tableau will match the aggregate rows based on linking fields.

*When you have more than one data source in a Tableau workbook, whichever source you use first in a view becomes the primary source for that view.*

*Blending is view-specific. You can have one data source as the primary in one view and the same data source as the secondary in another. Any data source can be used in a blend, but OLAP cubes, such as SSAS, must be used as the primary source.*

**Linking fields** are dimensions which are used to match data blended between primary and secondary data sources. Linking fields define the level of detail for the secondary source. Linking fields are automatically assigned if fields match by name and type between data sources. Otherwise, you can manually assign relationships between fields by selecting, from the menu, Data | Edit Relationships, as follows:

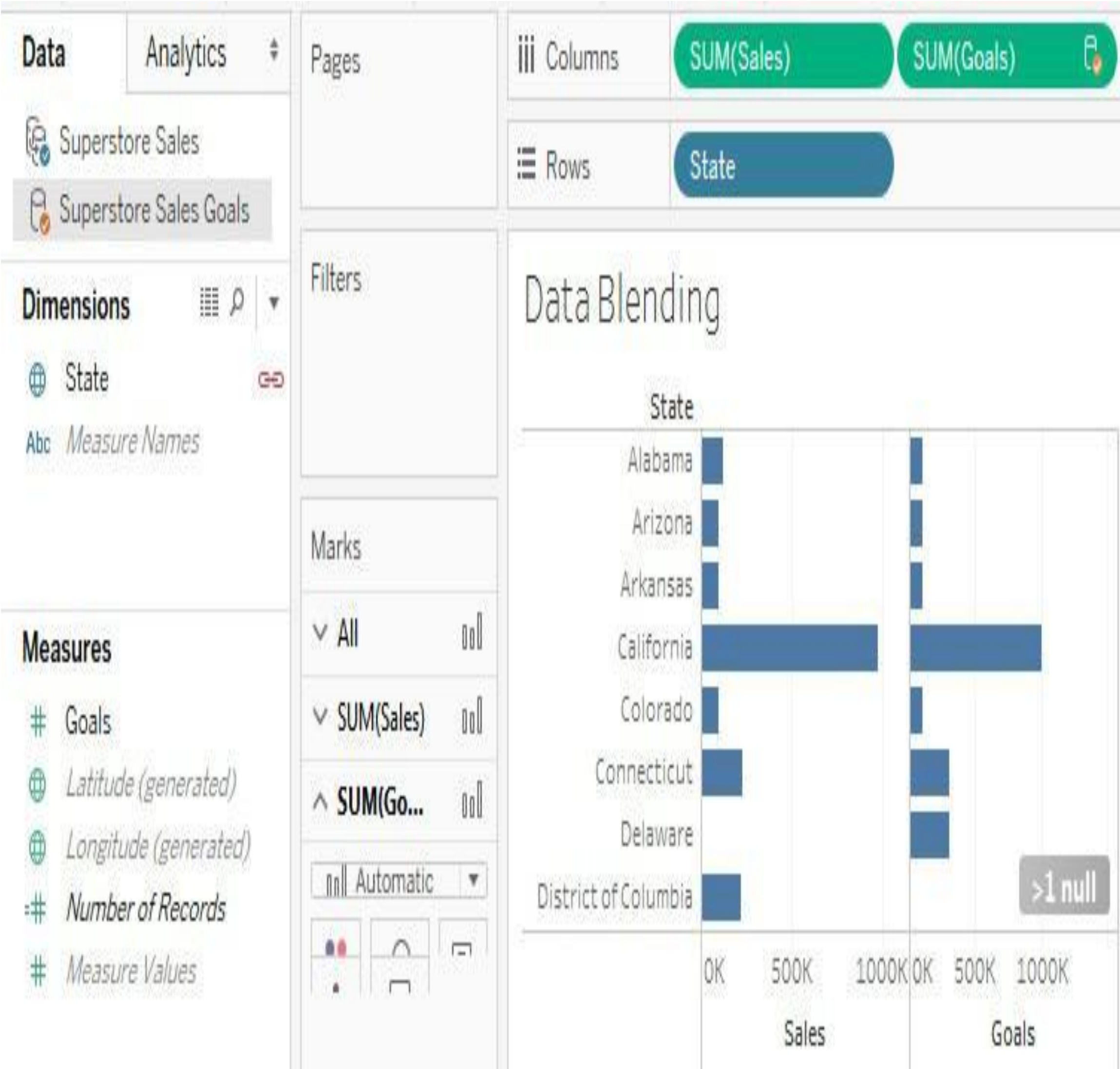
The Relationships window will display the relationships recognized between different data sources. You can switch from Automatic to Custom to define your own linking fields.

Linking fields can be activated or deactivated for blending in a view. Linking fields used in the view will usually be active by default, while other fields will not. You can, however, change whether a linking field is active or not by clicking the link icon next to a linking field in the Data pane.

*Additionally, use the Edit Data Relationships screen to define the fields that will be used for* **cross-data source filters***, which are discussed in the next section (Filtering).*

**A blending example**

The following screenshot shows a simple example of data blending in action:



There are two data source connections defined in this workbook, one for

the Superstore data and the other for Superstore Sales Goals. The Superstore data source is the primary data source in this view (indicated by the blue checkmark) and Superstore Sales Goals is the secondary source (indicated by

the orange checkmark). Active fields in the view that are from the secondary data source are also indicated with an orange checkmark icon.

The Sales measure has been used from the primary source and the Goals from the secondary sources. In both cases, the value is aggregated. The State dimension is an active linking field, indicated by the orange chain link icon next to the field in the data pane. Both measures are being aggregated at the level of State (sales by state in Superstore Sales, and goals by state in Superstore Sales Goals) and then matched by Tableau based on the value of the linking field State.

*Data blending will be done based on an exact match of the dimension values for the linking field(s). Be careful, as this can lead to some matches being missed. You'll notice the indicator in the lower-right of the preceding screenshot, which indicates > 1 null (at least one null value).*

*An examination of the data reveals that the State value in Superstore Sales is District of Columbia, while it is DC in the Superstore Sales Goals data source. You'll either need to fix the values in the data source, create a calculated field to change values, or change the alias of the value in one data source to match the value in another. For example, you could right-click District of Columbia in the row header, select Edit alias... and set the value to DC.*

We'll see some examples of editing aliases throughout the book. Take note of the following definition:

*An* **alias** *is an alternate value for a dimension value that will be used for display and data blending. Aliases for dimensions can be changed by right-clicking row headers or using the menu on the field in the view or in the Data Pane, and selecting the option for editing aliases.*

**Filtering data**

Often, you will want to filter data in Tableau in order to perform an analysis on a subset of data, narrow your focus, or drill into detail. Tableau offers multiple ways to filter data.

If you want to limit the scope of your analysis to a subset of data, you can filter the data at the source using one of the following techniques:

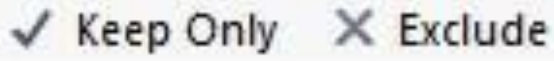
- **Data Source Filters** are applied before all other filters and are useful when you want to limit your analysis to a subset of data. These filters are applied before any other filters.
- **Extract Filters** limit the data that is stored in an extract (.tde or .hyper). Data source filters are often converted into extract filters if they are present when you extract the data.
- **Custom SQL Filters** can be accomplished using a live connection with custom SQL, which has a Tableau parameter in the WHERE clause. We'll examine parameters in Chapter 4, *Starting and Adventure with Calculations*.

Additionally, you can apply filters to one or more views using one of the following techniques:

Drag and drop fields from the data pane to the Filters shelf.

Select one or more marks or headers in a view and then select

Keep Only or Exclude, as shown here:



.

- Right-click any field in the data pane or in the view, and select Show Filter. The filter will be shown as a control (examples include a drop-down list and checkbox) to allow the end user of the view or dashboard the ability to change the filter.

- Use an action filter. We'll look more at filters and action filters in the context of dashboards.

Each of these options adds one or more fields to the Filters shelf of a view. When you drop a field on the Filters shelf, you will be prompted with options to define the filter. The filter options will differ most noticeably based on whether the field is discrete or continuous. Whether a field is filtered as a dimension or as a measure will greatly impact how the filter is applied and the results.

**Filtering discrete (blue) fields**

When you filter using a discrete field, you will be given options for selecting individual values to keep or exclude. For example, when you drop the discrete Department dimension onto the Filters shelf, Tableau will give you the following options:

Filter [Department]

General | Wildcard | Condition | Top

Select from list | Custom value list | Use all

Enter Text to Search

- [x] Furniture
- [ ] Office Supplies
- [x] Technology

All | None | Exclude

Summary

Field: [Department]

Selection: Selected 2 of 3 values

Wildcard: All

Condition: None

Limit: None

Reset | OK | Cancel | Apply

The Filter options include General, Wildcard, Condition, and Top tabs. Your Filter can include options from each tab. The Summary section on the General tab will show all options selected:

The General tab allows you to select items from a list (you can use the custom list add items manually if the dimension contains a large number of values that takes a long time to load.) You may

use the Exclude option to exclude the selected items.

- The Wildcard tab allows you to match string values that contain, start with, end with, or exactly match a given value.

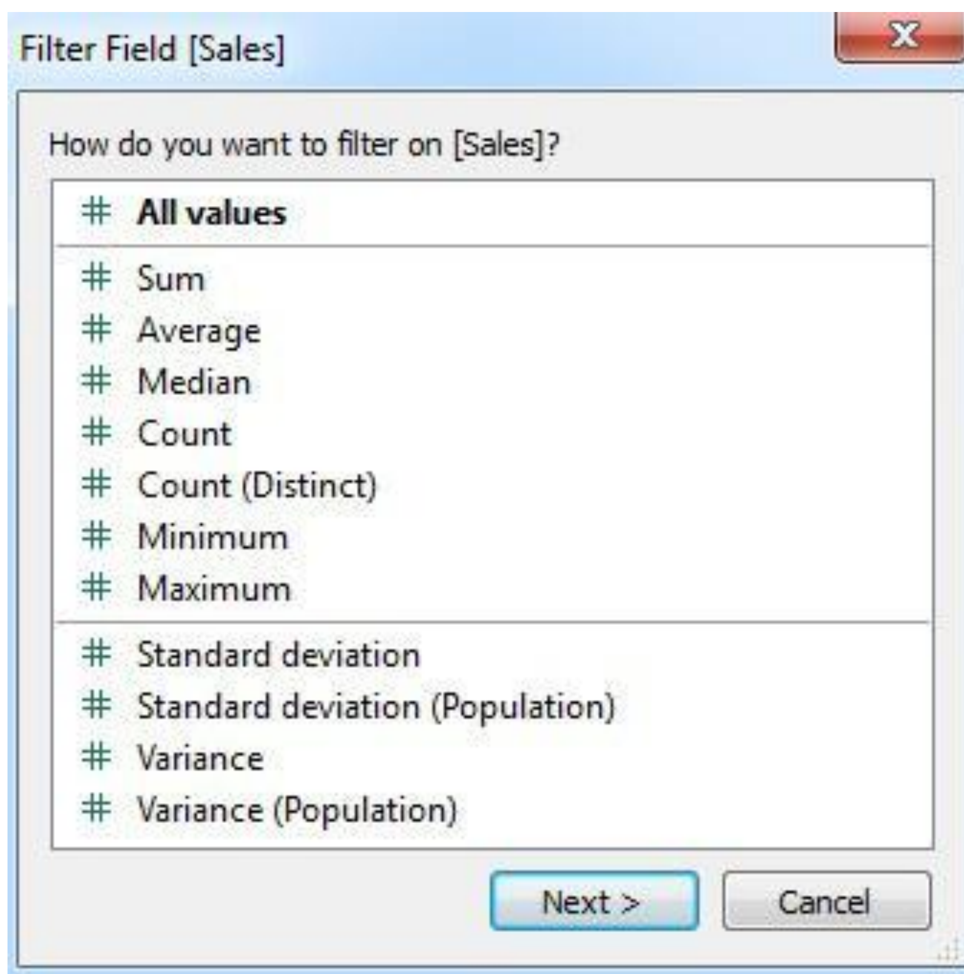
- The Condition tab allows you to specify conditions based on aggregations of other fields that meet conditions (for example, a condition to keep any Department where the sum of sales was greater than $1,000,000). Additionally, you can write a custom calculation to form complex conditions. We'll cover calculations more in Chapter 4, *Starting and Adventure with Calculations,* and Chapter 5, *Diving Deep with Table Calculations*.

- The Top tab allows you to limit the filter to only the top or bottom items. For example, you might decide to keep only the top five items by the sum of sales.

*Discrete measures (except for calculated fields using table calculations) cannot be added to the Filters shelf. If the field holds a date or numeric value, you can convert it to a continuous field before filtering. Other data types will require the creation of a calculated field to convert values you wish to filter into continuous numeric values.*

**Filtering continuous (green) fields**

If you drop a continuous dimension onto the Filters shelf, you'll get a different set of options. Often, you will first be prompted as to how you want to filter the field, as follows:



The options here are divided into two major categories:

- All Values: The filter will be based on each individual value of the

field. For example, an All Values filter keeping only sales above $100 will evaluate each record of underlying data and keep only individual sales above $100.

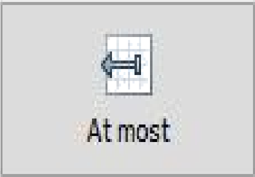
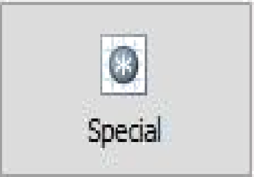
**Aggregation**: The filter will be based on the aggregation specified (for example, Sum, Average, Minimum, Maximum,

Standard deviation, and Variance) and the aggregation will be performed at the level of detail of the view. For example, a filter keeping only the sum of sales above $100,000 on a view at the level of category will keep only categories that had at least $100,000 in total sales.

Once you've made a selection (or if the selection wasn't applicable for the field selected), you will be given another interface for setting the actual filter, as follows:

Filter [Sales]

Range of values

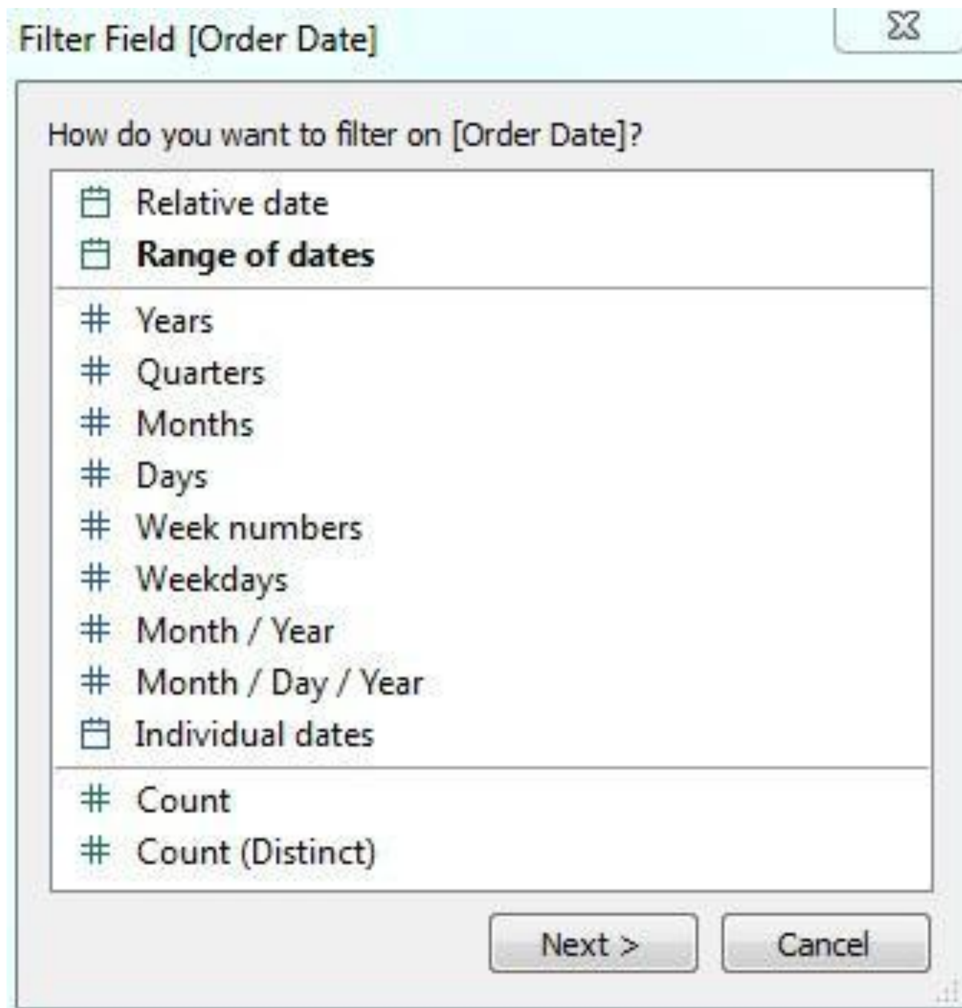
At least

At most

Special
At least

40,000
4,380,165.2
$32,249
$4,380,165
Show:
Only relevant values
Include null values
Reset
OK
Cancel
Apply

Here, you'll see options for filtering continuous values based on a range with a start, end, or both. The Special tab gives options for showing all values, NULL values, or non-NULL values.

### Filtering dates

We'll take a look at the special way Tableau handles dates in the *Visualizing Dates and Times* section of Chapter 3, *Venturing on to Advanced Visualizations*. For now, consider the options available when you drop an Order Date field onto the Filters shelf, as follows:



The options here include the following:

Relative date: This option allows you to filter a date based on a

specific date (for example, keeping the last three weeks from today, or the last six months from January 1).

- Range of dates: This option allows you to filter a date based on a range with a starting date, ending date, or both.

- **Date Part**: This option allows you to filter based on discrete parts of dates, such as Years, Months, Days or combinations of parts, such as Month/Year.

- Individual dates: This option allows you to filter based on each individual value of the date field in the data.

- Count or Count (Distinct): This option allows you to filter based on the count, or distinct count, of date values in the data.

**Other filtering options**

You will also want to be aware of the following options when it comes to filtering:

- You may display a filter control for nearly any field by right-clicking it and selecting Show Filter. The type of control depends on the type of field, whether it is discrete or continuous, and may be customized by using the little drop-down arrow at the upper-right of the filter control.

- Filters may be added to the **Context**. Any filters added to the context are evaluated first and result in what can be thought of as a subset of the data. Other filters and calculations (such as computed sets and fixed level of detail calculations) are based on the subset of data. This can be useful if, for example, you want to filter to the top five customers, but want to be able to first filter to a specific region. Making Region a context filter ensures that the top five filter is calculated in the context of the Region filter.

- Filters may be set to show all values in the database, all values in the context, all values in a hierarchy, or only values that are relevant based on other filters. These options are available via the drop-down menu on the Filter control.

- By default, any field placed on the Filters shelf defines a filter that is specific to the current view. However, you may specify the scope by using the menu for the field on Filters shelf. Select Apply to and choose one of the following options:

1. All related data sources: All data sources will be filtered by the value(s) specified. The relationships of fields are the same as blending (that is, the default by name and type match, or customized through the Data | Edit Relationships... menu option). All views using any of the related data sources will be affected by the filter. This option is sometimes referred to as **cross-data source filtering**.

2. Current Data Source: The data source for that field will be filtered. Any views using that data source will be affected by the filter.

3. Selected Worksheets: Any worksheets selected that use the data source of the field will be affected by the filter.

4. Current Worksheet: Only the current view will be affected by the filter.

When using Tableau Server, you may define user filters that allow you to provide row-level security by filtering based on user credentials.