Data Collection, Integration, and Analysis

Yatish Jain, Martin Schedlbauer and Kathleen Durant

Data Collection, Integration, and Analysis

Yatish Jain, Martin Schedlbauer and Kathleen Durant

This book is for sale at http://leanpub.com/collectingstoringandretrievingdata

This version was published on 2017-01-13

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2017 Dr. Martin Schedlbauer and Yatish Jain

Contents

Chapter 1 . . . . . **1**
Introduction 1
Big Data 2

Chapter 2 . . . . . **12**
Basic R Programming 12
Modes and Classes of objects 18
R Objects 23

Chapter 3 . . . . . **35**
More R Programming 35
Conditional Statements 42
Control structures 43

Chapter 4 . . . . . **53**
Data shaping 53

Chapter 5 . . . . . **67**
Algorithmic Complexity 67

Chapter 6 . . . . . **80**
Importing data in R 80

Chapter 7 . . . . . **94**
Web Scraping 94

Chapter 8 . . . . . **101**
HTTP Retrieval with HTML Parsing 101

Chapter 9 . . . . . **115**
Data collection through web APIs 115

Chapter 10 . . . . . **130**
Storage of data 130

Chapter 11 . . . . . **137**

Data insertion through R in SQLite 137

Chapter 12 . . . . **146**
Retrieving relational data 146

Chapter 13 . . . . **155**
Non-relational databases 155

Chapter 14 . . . . **161**
MongoDB 161
Neo4J 170

Chapter 15 . . . . **184**
Data Analysis 184

Chapter 16 . . . . **191**
Characterizing data through statistics 191

Chapter 17 . . . . **203**
Validating data through hypothesis testing 203

Chapter 1

Introduction

The value of data:

The fundamental unit of a civilization is people. People generate data through the processes they perform: historians record events, authors write books, musicians compose music and researchers generates studies. All of these processes performed by people generate data as a by-product. This is also true of corporations. In the process of building products, hiring employees, buying infrastructure, and polling customers, data is created as a by-product of these processes. A Savvy company can transform the data from a byproduct to an asset by using the data when making important decisions.

- Where to locate a new franchise
- What customers to target in marketing
- Where bottlenecks exist in a process
- How customers feel about a product

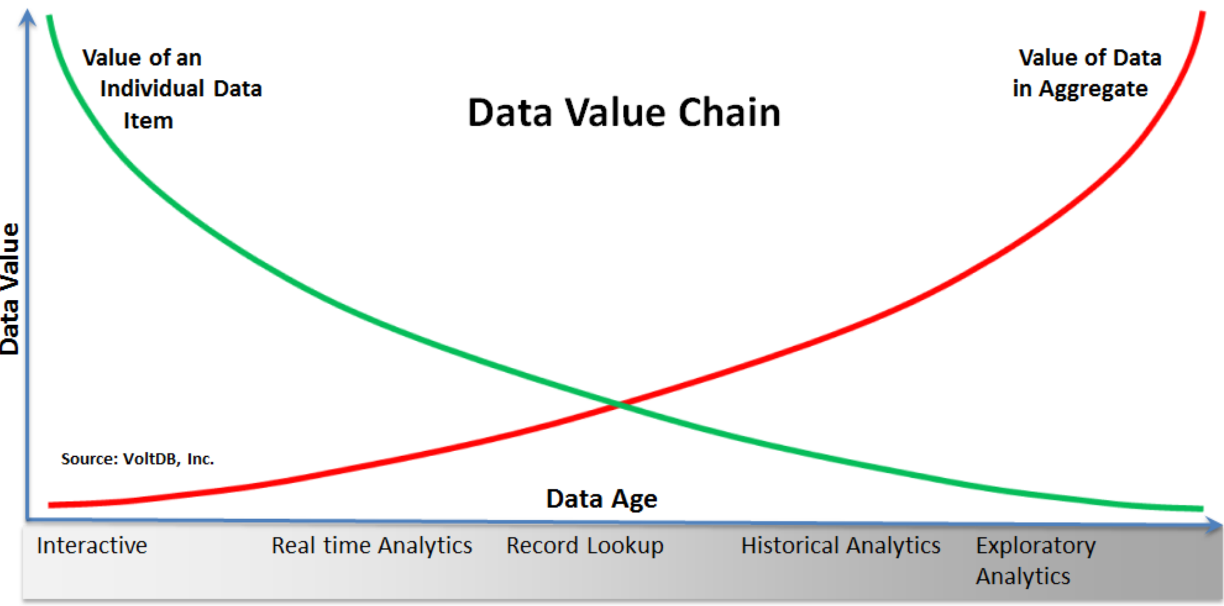


Fig. 1a- The value of data

Data needs to be in a format that allows it to be used for qualitative, quantitative and statistical analysis. In an ideal world, data is well organized, has no missing data values and is properly formatted. However, in the real world, data is often unformatted or formatted in a way that is not conducive to analysis. It may also be missing values for critical data variables, making it very difficult or sometimes impossible to perform the necessary analysis.

A	B	C	D	E	F
REGION	MARKET	STORE	IN BALANCE DATE	FISCAL PERIOD	MODEL
North	Great Lakes	65061011	01/03/03	200205	4055T
North	Shenandoah Valley	62067017	01/03/03	200205	2500P
North	Shenandoah Valley	32139049	01/03/03	200205	2500C
North	New England	2004014	01/03/03	200205	4055T
North	New England	72074014	01/03/03	200205	4500C
North	New England	12011011	01/03/03	200205	3002P
North	New England	2105015	01/03/03	200205	2500P
North	New England	22022012	01/03/03	200205	4055T
North	New England	22022012	01/03/03	200205	3002C

Fig. 1b Well formatted data


```
} ],
"metadata" : {
  "custom_fields" : {
    "TEST" : {
      "CFPB1" : ""
    }
  },
  "renderTypeConfig" : {
    "visible" : {
      "table" : true
    }
  },
  "availableDisplayTypes" : [ "table", "fatrow", "page" ],
  "rdfSubject" : "0",
  "rowIdentifier" : 53173967
},
"owner" : {
  "id" : "dfzt-mv86",
  "displayName" : "CFPB Administrator",
  "roleName" : "publisher",
  "screenName" : "CFPB Administrator",
  "rights" : [ "create_datasets", "edit_others_datasets", "e
"view_domain", "view_others_datasets", "create_pages", "edit_p
},
```


Fig. 1c Poorly formatted data with missing values

The Role of a Data Scientist

Data scientists extract actionable information from data collected from potentially many different sources. The top level data processes are listed below:

- Collecting the data in the raw form
- Data munging and data wrangling (the process of converting the original format to a more useful format) to make it useful for analysis and visualization
- Cleansing the data to deal with missing values and weirdly formatted data
- Curation of the data in order to make it available for reuse and preservation

Big Data

Big data is a relatively new term that describes datasets that are so large and complex that traditional data storage and processing methods can not be applied to these data sets. The following list are examples of big data generation happening every 60 seconds.

Every 60 seconds there are[1]:

- Facebook users share nearly 2.5 million pieces of content.
- Twitter users tweet nearly 300,000 times.
- Instagram users post nearly 220,000 new photos.
- Apple users download nearly 50,000 apps.
- Email users send over 200 million messages.
- Amazon generates over $80,000 in online sales.

The above statistics is just a glimpse into the voluminous, ever-growing collection of data that is available. The statistics below highlight why big data is a 21st century problem.

- Every day over 2.5 quintillion bytes of data is being generated.
- 90% of the world's data has been generated over the past two years.
- Data from multiple sources is being integrated into single massive data sets.

Due to the complexity involved with the term itself there is no single agreed upon definition of "Big Data", below is one definition that highlights the structure and the output of big data:

> **Big data** is the integration of large amounts of multiple types of structured and unstructured data into a single dataset that can be analyzed to gain insight and new understanding of an industry, business, the environment, medicine, disease control, science, and human interactions and expectations.

Examples of Big data:

- The Large Hadron Collider would generate 5 ×1020bytes per day if all of its sensors were turned on, almost 200 times more than all other data sources in the world combined.
- The Square Kilometer Array radio telescope is expected to collect 14 exabytes of data per day for analysis.
- Walmart generates over 1 million customer transactions per hour that are curated in a multi-petabyte database for trend analysis.

[1] https://aci.info/2014/07/12/the-data-explosion-in-2014-minute-by-minute-infographic/

Characteristics of Big Data:

Even though there is not one agreed upon definition of "Big Data", there are certain characteristics that allow you to identify big data.

- Very large, distributed aggregations of loosely structured data that is often incomplete
- In excess of multiple petabytes or exabytes of data
- Billions of records about people or transactions
- Loosely structured and often distributed data
- Flat schemas with few complex interrelationships
- Time series data containing time-stamped events
- Connections between data elements that must be probabilistically inferred through machine learning
- Security concerns of personal data
- Data can contradict itself over time and may have missing data values

Larger datasets allow for more detailed analysis and application to social sciences, biology, pharmacology, business, marketing and more. Data is everywhere and a lot of it is free. Organizations don't necessarily have to build their own massive data repositories before starting with big data analytics. Steps taken by many companies and government agencies to put large amounts of information into the public domain have made large volumes of data accessible to everyone.

Some of the important sources of data are:

Web Behavior and content:

- There are nearly five billion web pages, most of them are collecting data and statistics on its utility as well as its visitors
- The collected data includes network traffic, site and page visits, page navigation, page searches
- This data can be used for marketing purpose, such as generating advertisements based on your recent purchases

User Generated Content:

- Also known as "Internet trail" or "Net trail"
- Content generated by millions of users on social media, including Facebook, Twitter, Instagram, blogs, YouTube, forums, wikis, and so forth
- This data can an be used to create an online profile of any online user and the analysis of such data can be highly useful for targeting campaigns

Activity Generated data:

- Computer and mobile device log files
- Includes website tracking information, application logs, sensor data such as check-ins and location tracking
- This data can be used to generate location specific (geographic) marketing campaigns

RFID Data:

- Radio Frequency Identifiers
- These are tags for tracking merchandise, shipments, mobile payments, sports performance measurement, and automated toll collection
- This data can be used for tracking objects across the globe.

Geo-Data:

- GPS tracking data generated by mobile devices
- This data is another source for tracking the movement of equipment, vehicles, and people

Environmental Data:

- Weather conditions
- Traffic movements
- Tidal movements
- Seismic activity

Organizational Transactional Data:

- Transactional activities such as purchases, registration, manufacturing

Research Data:

- Social science data, e.g., census, polls
- Health care data
- Education, law and order, economic activity, agriculture, food production
- “Big Data” such as radio telescopes, particle physics

Big data is poised to offer tremendous insights, but with the terabytes and petabytes of data pouring into organizations today, traditional architectures are not up to the challenge to storing, collecting and analyzing this large amount of data. There are many challenges that come along with big data:

Analysis:

With the enormous amount of data available, the major challenge is to leverage the value that big data have to offer. Big data requires complex analysis within a relatively short time span, since it is used to detect and track trends, helping people make more informative decisions. Some analysis techniques applied to big data are:

- A/B Testing
- Information visualization
- Machine Learning Techniques
- Time Series Analysis

Collection:

- Data is not free, even when you do not pay for it. You still need to pay for the storage mechanism as well as the computational system used to analyze the data
- Data is in a format not conducive to analysis. Work needs to be expended to transform the data into an amenable format.
- Data contains missing values or bad entries. The quality of the data needs to be measured and protocol needs to be defined for dealing with invalid, inaccurate and missing data values.
- Data is not downloadable. Many websites provide mechanisms for interacting with the data records one by one, however gathering all data records is not provided. You will need to write code that scrapes the data from the website, if it is not provided as a simple file download.

Storage:

Storage of such enormous data is a challenge in itself. There is a need for the system to be able to deal with terabytes/petabytes of data on a daily basis. With big data a company must have a plan to deal with disk failures.

Curation:

Curation of data deals with addressing the quality of data. Data's penultimate value occurs when it is both timely and accurate. When data is timely and accurate it can assist corporate decision making processes. On the other hand, poor information quality can be costly. For example, an infographic from lemonly.com estimates that, on average, bad information costs businesses up to 10 - 25 percent of revenue per year. Same study pegs the loss at over $3 trillion annually in the U.S. alone.

Search and retrieval:

Timely retrieval of meaningful data from the entire dataset is one of the most important challenges.

Sharing/Transfer:

Sharing/Transferring data is another concern as there is no platform easily available which allows transfer of such huge data. Organizations tend to invest a lot of money to design special architectures and infrastructures to facilitate data sharing/transfer.

Visualization:

Visualization helps a user reason and analyze the data. It provides a mechanism for extracting useful information as well as a mechanism for defining an analysis plan.

Privacy:

Data security is a major concern especially when it comes to credit card data, personal ID information, health information or other sensitive assets. Protecting the privacy of valuable data can be a challenging feat. It requires both hardware and software solutions.

Storing Big Data:

Traditional data storage technologies including text files, XML, and relational databases reach their limits when used to store very large amounts of data. Furthermore, the data that is needed for analysis includes not only text and numeric data but unstructured data, such as text files, video, audio, blogs, sensor data, geospatial data among others. Due to these hurdles storing big data becomes challenging. A new crop of databases have been developed to fill the needs of big data, they are called NoSQL (Not Only SQL) databases. The collection of NoSQL databases allows data modelers to use other data models beyond the relational data model. The relational data model limits all data concepts to the two dimensional table, where the rows represent an entity and the columns represent an attribute or a feature of the entity. The NoSQL databases do not incorporate the two dimensional table model that relational database management systems (RDBMS) promote. NoSQL databases have the ability to deal with a large amount of data and can accommodate unstructured data easily. Fetching data from NoSQL databases provides remarkable speed over relational database since the data to answer a specific user question is stored to optimize specific user operations.

6 V's of Big Data

Volume

Volume is one of the core defining attributes of "Big Data". Big Data implies enormous amounts of structured and unstructured data that is generated by social and sensor networks, transaction and search history, and manual data collection. For example- 100 terabytes of data is uploaded daily to Facebook; Akamai analyzes 75 million events a day to target online ads; Walmart handles 1 million customer transactions every single hour. These are volumes that are new for the 21st century.

Variety

Data comes from a variety of sources and contains both structured and unstructured data. There are a variety of supported data types; big data is not restricted to simply numbers and short text fields, but may also include images, emails, text messages, web pages, blog entries, documents, audio, video, and time series.

Velocity

The flow of data that needs to be stored and analyzed is continuous, leading to a high rate or high velocity of data creation, data flow, and data consumption. Human interactions, business processes, machines, and networks generate data continuously and in enormous quantity. Here are some examples of big data consumption: every minute of every day, we upload 500 hours of video on Youtube[2], send over 204 million emails[3] and send 350,000 tweets[4]. Even though the data velocity date is high, it is important that the data is analyzed in real-time in order to gain a strategic advantage. Real time processing allows companies to do things like display personalized ads on the web pages you visit, that are based on your recent searches, web page viewing and purchase history. If the velocity rate is too high to allow real time processing, sampling can help mitigate some of the problems associated with large data volume and velocity.

Veracity

Data veracity characterizes the inherent noise, biases, abnormalities, and mistakes present in virtually all data streams. "Dirty data" presents a significant risk as analysis based on this data may be incorrect or misleading. Data must be cleaned in real-time and processes must be established to keep "dirty data" from accumulating. A data scientist needs to work as a "data janitor" before analyzing the data.

Validity

While the data may not be "dirty", biased, or abnormal, and it may not be valid for the intended use. Valid data for the intended use is essential to making decisions based on the data.

Volatility

Volatility characterizes the degree to which data changes over time. Decisions and analysis are based on data that has an "expiration date". Data scientists must define at what point in time a data stream is no longer relevant and cannot be used to make an informed decision.

[2]http://www.reelseo.com/hours-minute-uploaded-youtube/

[3]http://mashable.com/2014/04/23/data-online-every-minute/#77si5G_0YSqg

[4]http://www.internetlivestats.com/twitter-statistics/

Additional V's

Viscosity

Viscosity measures the data's resistance to flow through the data processes. It also can be used to measure the difficulty a user encounters when navigating to a specific data element within the dataset. Technologies to deal with viscosity include improved streaming, agile integration bus, and complex event processing.

Virality

Virality measures how quickly data is spread and shared to each unique node in the data network. Time is an important characteristic along with the rate of proliferation. High virality of data can provide companies with instant insights into the target population segments for a specific marketing campaigns.

Learning Checkpoint

Concepts Pharma has built a data repository that collects self-reported eating habits of clinical trial participants through a mobile habit. The translation medicine group is using the data to determine if the drug in the trial is causing digestive issues when taken with certain food groups. Which of the V's should be of most concern to them?

1. Veracity
2. Volume
3. Volatility
4. Velocity
5. Variety

Answer at the end of chapter

Planning A Big Data project

Developing a "Big Data" project requires thoughtful planning. The project must have clearly defined objectives and "questions" that need to be answered through analysis. The project plan must also address where the data will come from, the quality of the data, the processes for collecting, cleaning, and loading the data, and the infrastructure used to house the data. Finally, the project plan must state how the data is expected to be analyzed and how data will be kept free of identifiable properties and keep personal data confidential. A data scientist or data analyst, planning a big data project should address:

- Objectives
- Data
- Process
- Infrastructure
- Analytics
- Governance and Privacy

Objectives

Objectives need to be clearly defined with a proper outline of every step of the project. A data scientist needs to answer the questions like:

- What is the purpose of the data project?
- How is the data going to be used?
- What is the business or organizational value of the data project?

Data

When identifying the data, the following questions need to be answered:

- What data needs to be collected?
- Where will the data come from?
  - Internal systems?
  - Social networks?
  - External data sources?
- What is the structure of the data?
  - Quantitative or qualitative?
- What is the quality of the data?

Processes

When defining the data processes, the following questions need to be answered:

- How will the data be collected?
- How frequently will the data be collected?
- How will the data be cleaned?
- How will the data be loaded and transferred?
- What kind of analysis needs to be done?
- Will the system be able to provide real-time analysis?

Infrastructure

When choosing the system's infrastructure, the following questions need to be answered:

- Where will the data be stored?
- What database or data store will be needed based on the volume, complexity, type, and required access of the data?
- What hardware is needed to support responsive access to the data?
- Who will manage the data store?
- Who will supply the data store?

Analytics

When identifying the methods used to analyze the data, the following questions need to be answered:

- How will the data be presented?
  - Tables?
  - Visualizations?
- What predictive models will be built? How will these models be evaluated?
- How will the data from different sources be combined?
- What skills are needed to do the analysis?
- What programs or applications need to be built or purchased?

Governance and Privacy

When defining data access policies and availability a company must consider the following:

- Organizations must be transparent in how they manage personal data and how they use it.
- Government regulations may limit which data can be collected and how that data can be stored, transferred, accessed or used.
- Organizations must protect private data and not allow persons to be "identifiable".
- Organizations must follow governmental rules such as the data protection law, the Health Insurance Portability and Accountability Act (HIPAA),

Checkpoint Answer

Concepts Pharma has built a data repository that collects self-reported eating habits of clinical trial participants through a mobile habit. The translation medicine group is using the data to determine if the drug in the trial is causing digestive issues when taken with certain food groups. Which of the V's should be of most concern to them?

1. **Veracity**
2. Volume
3. **Volatility**
4. Velocity
5. Variety

Chapter 2

Basic R Programming

Before we start with the basics of R let's make sure you have the latest version of R. To install R on your computer go to the home website of R and follow the instructions there:

http://www.r-project.org/[5]

We recommend the use of Rstudio, a powerful IDE for R. Rstudio is also free and can be downloaded from their home website:

http://www.rstudio.com/[6]

Why R? When we deal with big data, we face many challenges, R provides a perfect platform to deal with these challenges. R provides a powerful environment which runs on several platforms, it can process an enormous amount of data using one statement or million chunks of data one by one. R also allows you to deal with bad or missing data and it makes reshaping and restructuring of data easy. Data manipulation becomes a lot easier with the different string and date manipulation packages. R provides packages that provide connectivity to many databases. R also provides many packages that provide statistical, graphical, and machine learning functionality. It also allows a program to call a function written in another programming language. This allows a programmer to take advantage of the prior work written in other programming languages within an R program. Many people leverage the programs they have written in Perl within an R program.

R objects

Data is stored as objects in R. Objects are created by:

- Reading data from an external file
- Retrieving data from a URL
- Creating an object directly from the command line
- Instantiating an object from within a program

[5]http://www.r-project.org/
[6]http://www.rstudio.com/

Expressions

R can be directly used to solve simple or complex expressions. The following is a log from the R shell. The '>' is the command prompt for the R shell. You can see you can type in arithmetic expressions and the shell evaluates them. You can also define variables as well as define character text strings. If you attempt to run any line of code from this book, you must remove this > symbol from your code. For example, line 1 would be entered as: 12*21. Also, the [1] expression in the log is a unique number for the records in the result set.

```
1   > 12*21
2   [1] 252
3
4   # [1] in the above answer indicates the index of your results.
5   # R always shows the result with index for each row.
6
7   > ((2^3)*5)-1
8   [1] 39
9
10  > sqrt(4)* exp(2)
11  [1] 14.77811
```

Note : sqrt and exp are built-in functions in R for finding Square root and exponential respectively.

Assignment

Assignment of a value to a variable can be done in 2 ways in R:

```
1   #first way
2   > x=12
3   > x
4   [1] 12
5
6   > word = "Hello"
7   > word
8   [1] "Hello"
9
10  #second way
11  > x <- 12
12  > x
13  [1] 12
14
```

```
> word <- "Hello"
> word
[1] "Hello"
```

The second method is more frequently used by R users.

The rules for naming an object are the following. Object names are case-sensitive and cannot contain spaces or special characters. An object identifier must start with a letter, but may contain any letter or digit thereafter.

It's a good coding practice to give a sensible name to a variable instead of just using random alphabet like x,y,z. Another good coding practice involves camelCasing a variable name i.e. instead of using an underscore to separate the two words that compose a name, make the first letter of the second word uppercase for example: squareRoot, graphData, currentWorkingDirectory.

Note that R is case sensitive which means that R treats the object names "AP" and "ap" as different objects. Accessing files is also most commonly case sensitive, so there's a difference between "AirPassengers.txt" and "airpassengers.txt".

Functions

R functions can be invoked by their name. Details of any built-in functions or dataset can be accessed by adding a question mark (?) in front of the function or dataset name.

```
> ?sum
```

sum {base} R Documentation

Sum of Vector Elements

Description

`sum` returns the sum of all the values present in its arguments.

Usage

`sum(..., na.rm = FALSE)`

Arguments

`...` numeric or complex or logical vectors.

`na.rm` logical. Should missing values (including `NaN`) be removed?

Fig 2a: Sum function

```
1  > sum(1,2,30)
2  [1] 33
3
4  > nums <- c(1,2,3,4,5,6,7)
5  > mean(nums)
6  [1] 4
7  #Concatenating numeric or character values using the built-in c()
8  #function results in an indexable array
```

User defined functions are an important part of programming in R. They allow code to be reused, we will discuss more user-defined functions at the end of this chapter.

```
1  > fraction<-function(x,y){  #function definition (this is a comment)
2  + result <-x/y              #function body
3  + print (result)            #function body
4  + }
5
6  # Plus sign is a part of the code in console once the code is executed
7  # which tells the computer that this code is in continuation with the above line.
8
9  # If you are writing this code to run do not write plus signs.
10
11 > fraction(3,2)             ##function call
12 [1] 1.5
```

Combine function

Combining numeric or character values using the built-in c() function results in an indexable 1 dimensional array. Operations can be applied to each element of an array by applying the operation to the array (see line 12 below in the R log session)

```
1  > array<-c(1,2,3,4,5,6,7,8)
2  > array
3  [1] 1 2 3 4 5 6 7 8
4  > array[2]
5  [1] 2
6
7  > array + 10
8  [1] 11 12 13 14 15 16 17 18
9  > array
10 [1] 1 2 3 4 5 6 7 8
11
```

```
12  > fraction <- array/2
13  > fraction
14  [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0
15
16  > sum <- fraction + array
17  > sum
18  [1]  1.5  3.0  4.5  6.0  7.5  9.0 10.5 12.0
```

Sequences and subscripting

Sequencing or range of numbers can be selected in R using ":" operator.

```
1   > 1:10
2   [1] 1 2 3 4 5 6 7 8 9 10
3
4   > 5:12
5   [1] 5 6 7 8 9 10 11 12
6
7   > 3:-3
8   [1] 3 2 1 0 -1 -2 -3
9
10  > 2*1:5
11  [1] 2 4 6 8 10
12
13  > 2*(1:5)
14  [1] 2 4 6 8 10
15
16  > array<-c(1:25)
17  > array
18   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Advanced sequencing can be done using built-in R function called seq(). seq() function is internally creating a data object, an object that does not have a name. It is stored in .Last.value object.

```
1   # Increment by 3
2   > seq(from=5,to=15,by=3)
3  [1] 5 8 11 14
4
5  # divide in 6 parts
6  > seq(from=1,to=10,length=6)
7  [1] 1.0 2.8 4.6 6.4 8.2 10.0
8
9  # divide in 4 parts with decrement of 2.5
10 > seq(from=100,length=4,by=-2.5)
11 [1] 100.0 97.5 95.0 92.5
12
13 # divide in parts equal to the vector range
14 > x <-10:20
15 > seq(from=50,to=52,along=x)
16 [1] 50.0 50.2 50.4 50.6 50.8 51.0 51.2 51.4 51.6 51.8 52.0
```

Sequences are vectors that are essentially 1 dimensional arrays and particular elements of a sequence can be extracted with the [] subscript operator. Subscripting in R is more flexible than many other programming languages.

```
1 > # extract the 3rd element
2 > x[3]
3 [1] 12
4
5 > # extract all BUT the 3rd element
6 > x[-3]
7 [1] 10 11 13 14 15 16 17 18 19 20
```

combine function c() can be used in conjunction with sequencing to retrieve a subset of elements.

```
1  > arr<-c(10:20)
2  > arr
3  [1] 10 11 12 13 14 15 16 17 18 19 20
4
5  #retrieve the 5th and 7th elements
6  > arr[c(5,7)]
7  [1] 14 16
8
9  #retrieve all but the 3rd, 5th, and 9th elements
10 > arr[c(-3,-5,-9)]
11 [1] 10 11 13 15 16 17 19 20
```

Specific elements that meet a logical criterion can be selected using subscripting.

```
> arr
[1] 10 11 12 13 14 15 16 17 18 19 20

#extract all elements greater than 14
> arr[arr>14] #returns the values within a list that satisfies a condition
[1] 15 16 17 18 19 20

> logical<-c(T,T,F,T,F,T,F)
> logical
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE

> logical[logical==T]
[1] TRUE TRUE TRUE TRUE

> which(logical==T)
[1] 1 2 4 6
# Enclosing this condition in a which function will return the
# index of satisfied condition
```

Listing and Deleting objects

The great strength of R lies in the flexibility it provides. The R shell overloads the definition of the ls() and rm() functions from the Unix shell command.

```
> ls()
[1] "a"    "b"    "c"    "r"    "word" "x"    "z"
> rm("word")
> ls()
[1] "a" "b" "c" "r" "x" "z"
#remove all objects  from current session
rm(list=ls())
```

Comments

Comments are an important part of any program/code. R code should be commented so that you or others understand the intent of the commands and functions. Any text after a hash mark (#) is a comment in the code and is ignored by R.

Modes and Classes of objects

Everything in R is stored in a data object. An R object has a mode and a class. The mode represents type of values that can be stored in it. The class represents the structure of the object.

Object	Description	Example Values
Constant	a numeric value	8, -3, 3.14
Text	a string of characters	"CS6020"
NULL	null reference	NULL
NA	missing value	NA
NaN	not a number	NaN
Inf	infinity	Inf

Fig. 2b Storage mechanism in R

There are specific rules for naming an object/variable in R:

- Variable names must start with a letter followed by upper and lower case letters, digits, period, and underscore (_)
- The following characters are not allowed in a variable name. characters: #, @, &, %, ^, $, ~, *

Variable names examples: rangeValue, i3, open_date

Every R object is a list or a vector. A vector contains elements that have the same mode. A list allows the data elements to have different modes. A list object has a class = 'list'. A list is considered a recursive data object since an element in a list may be a list.

- Mode -Numeric, character, logical
- Class

Mode

The mode is defined in R as a mutually exclusive classification of objects according to their basic structure. An object has only one mode. The values of mode are:

- numeric
- complex
- logical
- character
- raw

The mode() function returns the mode of an object. The modes of two combining objects must match when they are combined in an operation.

```
1  > num.vec<- c(1,2,3,4,5)
2  > mode(num.vec)
3  [1] "numeric"
4
5  > logical.vec<- c(T,F,F,T,F,T,T)
6  > mode(logical.vec)
7  [1] "logical"
8
9  > char.vec<- c("AA","BB","cc","dd")
10 > mode(char.vec)
11 [1] "character"
```

R follows specific rules for determining the mode of a combined object. It chooses the most accommodating data type when determining the object's mode. For example, R stores numeric objects as either 32-bit integers or double-precision floating point numbers. If an R object contains both numeric and logical elements, the mode of the objects is numeric and all logical elements are converted to numeric values with TRUE = 1 and FALSE = 0. If an R object contains character and numeric or logical elements, it is converted to a character mode. See the log below for an example.

```
1  > nl<-c(1,3,TRUE,9,FALSE)
2  > nl
3  [1] 1 3 1 9 0
4
5  > mode(nl)
6  [1] "numeric"
7
8  > cnl<-c(1,TRUE,"x")
9  > cnl
10 [1] "1" "TRUE" "x"
11
12 > mode(cnl)
13 [1] "character"
```

The mode of any object can be determined by is.modeName() function like is.numeric() is.character(). These functions return True or False as output. Functions are also treated as objects in R.

```
1  > is.numeric(num.vec)
2  [1] TRUE
3
4  > is.numeric(logical.vec)
5  [1] FALSE
6
7  > is.logical(logical.vec)
8  [1] TRUE
9
10 > mode(mean)
11 [1] "function"
12
13 > mode(sum)
14 [1] "function"
```

Class

The class of an object determines what can be done with the object, while the mode indicates the values that can be stored in the memory location. The class is accessed through the class() function.

If an object has no class assigned to it, then by default, its class is considered the same as the mode of that object.

```
1  > class(mean)
2  [1] "function"
3
4  > class(num.vec)
5  [1] "numeric"
6
7  > x<- 1:16
8  > x
9   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
10
11 > mode(x)
12 [1] "numeric"
13 > class(x)
14 [1] "integer"
15
16 > dim(x)<- c(4,4)
17 > x
18      [,1] [,2] [,3] [,4]
19 [1,]    1    5    9   13
20 [2,]    2    6   10   14
```

```
21 [3,]    3    7   11   15
22 [4,]    4    8   12   16
23
24 > mode(x)
25 [1] "numeric"
26
27 > class(x)
28 [1] "matrix"
```

The mode of an object can be changed using mode conversion functions and the process is called “coercion”. While some coercions are automatic, specific coercions can be forced using the family of as.mode() functions:

- as. numeric()
- as.logical()
- as.character()
- as.data.frame()
- as.list()

```
1  > n.vec<-c(-99,0,99)
2  > l.vec<-as.logical(n.vec)
3
4  > n.vec
5  [1] -99 0 99
6
7  > l.vec
8  [1] TRUE FALSE TRUE
9
10 > mode(l.vec)
11 [1] "logical"
```

Not all values can be coerced from one data type to another, when a value cannot be coerced, it is replaced with NA (not available).

```
> char.vec<- c("1","2","3","dd")
> mode(char.vec)
[1] "character"

> num<-as.numeric(char.vec)
Warning message:
NAs introduced by coercion
> num
[1]  1  2  3 NA
# R cannot convert "dd" to any number on its own and hence NA is introduced
```

R Objects

The following is a list of the R data objects:

- Vector
- Factors
- Dataframes
- Matrices
- Arrays
- Lists

Vector

The vector is the basic storage type in R. A vector holds a collection of values of the same type: numeric, logical, character. A vector is treated like an array and its elements can be accessed by indexing.

```
> num.vec<- c(1,2,3,4,5)
> num.vec
[1] 1 2 3 4 5

> logical.vec<- c(T,F,F,T,F,T,T)
> logical.vec
[1]  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE

> char.vec<- c("AA","BB","cc","dd")
> char.vec
[1] "AA" "BB" "cc" "dd"
```

Factor

The factor type is used to encode categorical data values. While a vector can have any number of distinct elements, a factor value is limited to its categories. Factors are essential for certain statistical hypothesis tests and models. Factors are stored as numbers internally.

Factors are created using the factor() function which requires a vector of category values as input.

```
1  > week <- c("Monday","Tuesday","Wednesday","Thursday",
2  + "Friday","Saturday","Sunday")
3  > fac <- factor(week)
4  > fac
5  [1] Monday    Tuesday   Wednesday Thursday  Friday    Saturday  Sunday
6  Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday
7
8  # When you create a factor from a data object you can provide a default
9  # ordering for the values. If you do not provide an ordering it will use
10 # the canonical ordering for the data type so ascending order for numeric
11 # data and alphabetical ordering for character data. Since you probably do
12 # not want to order the days of the week alphabetically, here is an example
13 # of specifying the ordering sequence with the factor function
14
15 > fac2 <- factor(week,levels = c("Monday","Tuesday","Wednesday",
16 + "Thursday","Friday","Saturday","Sunday"),ordered=TRUE)
17
18
19 > labels(fac)
20 [1] "1" "2" "3" "4" "5" "6" "7"
21
22 > levels(fac)
23 [1] "Friday"    "Monday"    "Saturday"  "Sunday"    "Thursday"  "Tuesday"
24 [7] "Wednesday"
25
26 > levels(fac2)
27 [1] "Monday"    "Tuesday"   "Wednesday" "Thursday"  "Friday"    "Saturday"
28 [7] "Sunday"
```

While factors are stored as unique numeric labels they are not, in fact, of numeric type. Therefore, factors cannot be used in numeric operations.

```
> mean(fac)
[1] NA
Warning message:
In mean.default(fac) : argument is not numeric or logical: returning NA
```

Adding or dropping levels in a factor variable, can be tricky. We cannot directly add a new entry to a factor like indexed vector. We will have to first add a new level to a factor variable and then we can add labels as an indexed vector for that particular level.

```
# Wrong method. Level Noday doesn't exist

> fac[8] <- "Noday"
Warning message:
In `[<-.factor`(`*tmp*`, 8, value = "Noday") :
  invalid factor level, NA generated

> fac
[1] Monday    Tuesday   Wednesday Thursday  Friday    Saturday  Sunday   <NA>
Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday


# Correct way. Make a level Noday first.

> fac<- factor(fac, levels = c(levels(fac), "Noday"))
> fac[8] <- "Noday"
> fac
[1] Monday    Tuesday   Wednesday Thursday  Friday    Saturday  Sunday Noday
Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday Noday

> fac[9]<- "Noday"
> fac
[1] Monday    Tuesday   Wednesday Thursday  Friday   Saturday  Sunday   Noday
[9] Noday
Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday Noday
```

Dropping levels can be done similarly by just removing the levels and refactoring the levels.

```
1  > fac <- fac[fac != "Noday"]
2  > fac
3  [1] Monday    Tuesday   Wednesday Thursday  Friday    Saturday  Sunday
4  Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday Noday
5
6  #Refactoring the levels. if there is no value for Noday then,
7  #refactoring will remove that label
8  > new.fac<- factor(fac)
9  > new.fac
10 [1] Monday    Tuesday   Wednesday Thursday  Friday    Saturday  Sunday
11 Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday
```

Data Frame

A data frame is a tabular arrangement of rows and columns. The columns are vectors and/or factors, and are similar to the layout of a spreadsheet. The columns represent data attributes while the rows represent records with values for the attributes. Two or more vectors can be combined to make a data frame given that they are of the same length. A data frame is created with the data.frame() function requiring the different vectors as input. The input vectors represent the columns of the tabular arrangement.

```
1  > var1 <-c("BOS","EWR","PBI","CVG") #creates a vector with 4 string elements
2  > var2 <-c(14,19,0,12) #creates a vector with 4 numeric element
3  > var3 <-factor(c("Above","Above","Normal","Below"))
4  > snow.frame<-data.frame(var1,var2,var3)
5
6  #columns must have same length
7  > snow.frame
8  var1 var2 var3
9  1 BOS 14 Above
10 2 EWR 19 Above
11 3 PBI 0 Normal
12 4CVG 12 Below
```

Columns can be directly accessed using the $ as the column operator: frame$col. The number prefixed on each row are not part of the data frame but part of the output. It allows you to identify the beginning of each data row.

```
1  > snow.frame$var1
2  [1] BOS EWR PBI CVG
3  Levels: BOS CVG EWR PBI
4  > snow.frame$var2
5  [1] 14 19 0 12
6
7  # Different ways of accessing columns
8  > snow.frame$var1
9  [1] BOS EWR PBI CVG
10 Levels: BOS CVG EWR PBI
11
12 > snow.frame[["var1"]]
13 [1] BOS EWR PBI CVG
14 Levels: BOS CVG EWR PBI
15
16 > snow.frame[,1]
17 [1] BOS EWR PBI CVG
18 Levels: BOS CVG EWR PBI
```

Matrices

A matrix is a two-dimensional arrangement of data similar to a data frame but unlike a data frame its elements must be of the *same data type*. To perform mathematical operations on matrices, its elements must be numeric. When a matrix is printed the output contains the name of the columns in the first row. Each row element is prefixed with a row number.

```
1  > mat<- matrix(c(1:10), nrow=5, ncol=4, byrow= TRUE)
2  > mat
3       [,1] [,2] [,3] [,4]
4  [1,]    1    2    3    4
5  [2,]    5    6    7    8
6  [3,]    9   10    1    2
7  [4,]    3    4    5    6
8  [5,]    7    8    9   10
```

The function **t()** creates a transpose of a matrix by interchanging its columns and rows.

```
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10    1    2
[4,]    3    4    5    6
[5,]    7    8    9   10
> t(mat)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9    3    7
[2,]    2    6   10    4    8
[3,]    3    7    1    5    9
[4,]    4    8    2    6   10
```

Two matrices can be multiplied using the `%*%` matrix multiplication operator.

```
> t(mat)%*%mat
     [,1] [,2] [,3] [,4]
[1,]  165  190  125  150
[2,]  190  220  150  180
[3,]  125  150  165  190
[4,]  150  180  190  220
```

Arrays

An array is a multi-dimensional data structure, while matrices and data frames are two-dimensional row/column arrangements ;a vector is a single dimension data structure. The log below creates a three dimensional array called twoDArray. Notice the way the three dimensional object is printed. It prints the first two by two array, followed by the second two by two array, followed by the third two by two array.

```
> twoDArray <-array(dim=c(2,2,3))
> twoDArray[,,1] <-rnorm(2)
> twoDArray[,,2] <-rnorm(2)
> twoDArray[,,3] <-rnorm(2)

> twoDArray
, , 1
[,1] [,2]
[1,] 0.5370590 0.5370590
[2,] 0.5897154 0.5897154
, , 2
```

```
12 [,1] [,2]
13 [1,] 0.86947620 0.86947620
14 [2,] -0.05387594 -0.05387594
15 , , 3
16 [,1] [,2]
17 [1,] -1.4424228 -1.4424228
18 [2,] -0.9510549 -0.9510549
```

Array elements are accessed through subscripting:

```
1 > a[1,1,1]
2 [1] 0.537059
3 > a[1,1,2]
4 [1] 0.8694762
```

Lists

A list object is a generic collection that can store objects of any type, including vectors, matrices, arrays, and data frames. This is essentially a bag data structure.

```
1 > l<- list(var1,snow.frame,mat,a)
2 > l
3 [[1]]
4 [1] "BOS" "EWR" "PBI" "CVG"
5
6 [[2]]
7   var1 var2   var3
8 1  BOS   14  Above
9 2  EWR   19  Above
10 3  PBI    0 Normal
11 4  CVG   12  Below
12
13 [[3]]
14      [,1] [,2] [,3] [,4]
15 [1,]    1    2    3    4
16 [2,]    5    6    7    8
17 [3,]    9   10    1    2
18 [4,]    3    4    5    6
19 [5,]    7    8    9   10
20
21 [[4]]
22 , , 1
```

```
23
24            [,1]       [,2]
25 [1,] -1.1241404 -1.1241404
26 [2,] -0.5584957 -0.5584957
27
28 , , 2
29
30           [,1]      [,2]
31 [1,] -1.272635 -1.272635
32 [2,]  1.272189  1.272189
33
34 , , 3
35
36           [,1]      [,2]
37 [1,] -1.729171 -1.729171
38 [2,] -0.412405 -0.412405
39
40
41 #accessing elements
42
43 > l[[3]]
44      [,1] [,2] [,3] [,4]
45 [1,]    1    2    3    4
46 [2,]    5    6    7    8
47 [3,]    9   10    1    2
48 [4,]    3    4    5    6
49 [5,]    7    8    9   10
50 > l[3]
51 [[1]]
52      [,1] [,2] [,3] [,4]
53 [1,]    1    2    3    4
54 [2,]    5    6    7    8
55 [3,]    9   10    1    2
56 [4,]    3    4    5    6
57 [5,]    7    8    9   10
```

The code below shows modification of a list variable, by directly accessing an individual element of the list variable and making the appropriate modifications.

```
1  > l[[1]][1] <- "MA"
2  > l[[1]]
3  [1] "MA"  "EWR" "PBI" "CVG"
```

User-defined functions

R is a procedural programming language. It provides many built-in functions as illustrated in this chapter. User-defined functions allow a programmer to reuse code within a program. They are the building blocks of a well defined program. Functions modularize your program. Functions typically have three parts: a function definition, a function body and a function call. The following line 1 through 8 defines a function called getStats. Line 11 is an example of calling the function getStats.

```
1  > getStats<- function(data) #function definition
2  { # Start of function body
3  +   meanData<-mean(data)
4  +   stdDev<-sd(data)
5  +   medianData<-median(data)
6  +
7  +   return(c(meanData,stdDev,medianData))
8  + } # end of function body
9
10 > data<-c(12,34,54,65,25,75,90,23,12,45,65,76)
11 > stats<-getStats(data) #function call
12 > stats
13 [1] 48.00000 26.70036 49.50000
```

In the above code, the function definition is the section of the code where the function is named and the arguments are declared. i.e. the first line of the function. According to this function definition, the function takes one argument during a function call.

The function body is the general logic or segment of code inside the function. The segment of code inside the function has a local scope, this means any variables declared within the function are only know within the function. The local variables do not exist outside this function. One important part of the function body is the return statement. Typically, you will want to assign the function to a variable. Sometimes functions will also produce output to the console. This print behavior is typically called a side effect of the function. Whatever we want to do via a function can be done in two ways: either print the result inside the function or return the values using the return statement which will be stored in a variable where the function was called. If you return the values to the calling code then that code has access to these values. If you only print the values then the calling code does not have access to the data object.

```
1  > getStats<- function(data){
2  +   meanData<-mean(data)
3  +   stdDev<-sd(data)
4  +   medianData<-median(data)
5  +
6  +   print(c(meanData,stdDev,medianData))
7  + }
8  >
9  > data<-c(12,34,54,65,25,75,90,23,12,45,65,76)
10 > getStats(data)
11 [1] 48.00000 26.70036 49.50000
```

NOTE - In the example above, there is no return line in this version of the getStats function, so we did not store the return value of getStats in a variable.

A function call is an instantiation of the function. This means the function call is run at the line where the function call is made. Function parameters allow you to pass data to the function. When calling a function, you specify a value for each argument as specified in the function's declaration. The arguments passed to a function can be of any datatype, however the code body will only work with the specific datatype. It is good practice to list the data types for the arguments within the comments.

```
1  > helloMessage<- function(name){ #this function takes name as argument.
2  +   if(is.character(name)==TRUE){
3  +     print(paste("Hello",name))
4  +   }
5  +   else{
6  +     print("Error")
7  +   }
8  + }
9
10 > helloMessage("Michelle")
11 [1] "Hello Michelle"
12 > helloMessage(123)
13 [1] "Error"
```

Global vs local variables in functions

Global variables are those variables which have global scope in a program. A global scope means the variable is known throughout the complete program and can be used anywhere in the program. A variable with a local scope, is restricted to the function or block of code where it is defined.

```
1  > count=0 #global scope
2  > increasingCountBy5 <- function(ctr){ #function definition
3  + ctr=ctr+5 #local scope limited to this function
4  + counter=count # bad practice using a global variable inside function
5  + counter=counter+1 #local scope limited to this function
6  + return(paste("Counter =",counter,"ctr =",ctr,"count =",count))
7  + }
8
9  > increasingCountBy5(count) #function call statement
10 [1] "Counter = 1 ctr = 5 count = 0"
11
12 > print(counter)
13 Error in print(counter) : object 'counter' not found
14 > print(ctr)
15 Error in print(ctr) : object 'ctr' not found
```

In the above code, count is the only global variable. ctr is the variable passed as an argument to the function and counter is a local variable defined inside the function. We can pass the global variable count to the function increasingCountBy5; when the function call statement is executed the passed variable's value is passed to the function as the variable ctr. The changes made to the argument will not be reflected in the global variable count, in other words changes made to the argument will not be reflected outside of the function.

> **NOTE** - Make a strict habit of not using a global variable inside the function directly! It is a poor programming practice. If you want your function to access some variable then you should always pass that variable as an argument to that function.

In the above code you can also notice that variable ctr and counter are not present outside the function, i.e. their scope is limited to that function where they are defined.

Why use functions

- **Code Organization** - Writing short functions to perform specific, well-defined tasks helps structure the code of a program.
- **Testing and debugging** - If you make functions, then it is easier to pinpoint problems within the code. Testing individual functions is easier, when you pass an argument that you know the expected result.
- **Removing redundancy** - Functions help in removing code redundancy. If you are writing the same block of code twice in any program, then that block of code should be a function.
- **Faster development time** - By organizing the code in functions, development time is shortened since there is less code to write. Functions make the code easier to understand and improves the organization of the program.

- **Easier maintenance** - If you revisit your code after a period of time and you organized your code using functions, the modularity of the functions makes it easier to understand and edit the code.

Chapter 3

More R Programming

Data Frames

Data frames are the most common type of compound data structures used in R in addition to scalar values (vectors) and collections of values (array sequences). They are similar to C++ and Java objects or C's arrays of data structures. A data frame is composed of multiple values each of which is commonly a sequence.

A data frame is often created by loading data from an external file or created internally. Data frames are essentially spreadsheets of columns and rows.

```
1  > seq1<-1:10
2  > seq2<-seq(from=100,to=300,by=5)
3
4  # create a new data frame 'df'
5  > df<-data.frame(seq1,seq2)
6  Error in data.frame(x, y) :
7  arguments imply differing number of rows: 10, 41
8  > seq2<-seq(from=100,to=300,length=10)
9  > df<-data.frame(seq1,seq2)
```

Note: To combine two vectors into a data frame they have to be of the same length

Individual elements of a data frame can be accessed the same way individual elements of an array, by using [] subscript operator.

```
1  > df
2      x        y
3  1   1 100.0000
4  2   2 122.2222
5  3   3 144.4444
6  4   4 166.6667
7  5   5 188.8889
8  6   6 211.1111
9  7   7 233.3333
```

```
10  8   8 255.5556
11  9   9 277.7778
12  10 10 300.0000
13
14  > df[6,2]
15  [1] 211.1111
16
17  # Accessing entire column
18  > df[,2]
19   [1] 100.0000 122.2222 144.4444 166.6667 188.8889 211.1111 233.3333 255.5556 277\
20  .7778 300.0000
21
22  # Accessing entire row
23  > df[1,]
24    x   y
25  1 1 100
```

Details about what any variable is storing can be determined using the structure function str(). Other functions used with a data frame are; dim() to determine the dimensions of any variable, length() to determine the length of the data frame, ncol() to determine the number of columns in the data frame and nrow to determine the number of rows in the data frame.

```
1   > str(df)
2   'data.frame':        10 obs. of  2 variables:
3    $ x: int  1 2 3 4 5 6 7 8 9 10
4    $ y: num  100 122 144 167 189 ...
5
6   > ncol(df)
7   [1] 2
8   > nrow(df)
9   [1] 10
10
11  > length(df)
12  [1] 2
13  > dim(df)
14  [1] 10  2
15
16  > length(df$x)
17  [1] 10
18  > dim(df)[1]
19  [1] 10
20
```

```
21 #referencing the last element
22 > x
23 [1] 1 2 3 4 5 6 7 8 9 10
24 > x[length(x)]
25 [1] 10
```

R comes with many built-in datasets which can be used for practice purpose. A complete list of built-in datasets can be accessed using the homepage of package datasets. Datasets Package[7]

The “discoveries” dataset contains the numbers of “great” inventions and scientific discoveries in each year from 1860 to 1959.

```
1  > discoveries
2  Time Series:
3  Start = 1860
4  End = 1959
5  Frequency = 1
6    [1]  5  3  0  2  0  3  2  3  6  1  2  1  2  1  3  3  3  5  2  4  4  0  2  3  7\
7   12  3 10  9  2  3  7
8   [33]  7  2  3  3  6  2  4  3  5  2  2  4  0  4  2  5  2  3  3  6  5  8  3  6  6\
9    0  5  2  2  2  6  3
10  [65]  4  4  2  2  4  7  5  3  3  0  2  2  2  1  3  4  2  2  1  1  1  2  1  4  4\
11   3  2  1  4  1  1  1
12  [97]  0  0  2  0
13
14 #converting in built data into a dataframe
15 > Discoveries<- data.frame(year=1860:1959,count=discoveries)
16 > head(Discoveries)
17   year count
18 1 1860     5
19 2 1861     3
20 3 1862     0
21 4 1863     2
22 5 1864     0
23 6 1865     3
```

The head() and tail() functions list the first and last six rows of a data frame respectively. These functions come in handy when dealing with larger datasets.

Given a logical statement, any() tests if at least one value in the set meets a criterion. This is a useful function for querying values stored in a data structure.

[7] https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html

```
1  > any(Discoveries[,2]<0)
2  [1] FALSE
3
4  > any(Discoveries[,1] < 1860 | Discoveries[,1] > 1959)
5  [1] FALSE
6  # A logical statement can consist of more than 1 clause.
7  # The '|' represents the OR of the 2 conditionals '&' is the and
8  # of the two conditionals. In this example, we are testing to see
9  # if there are any years that are less than 1860 or greater than 1959.
```

Descriptive statistics

Within this book, we will be mainly dealing with big data through R. In order to get a sense of your data, statistical functions such as mean(), max(), which(),are helpful in navigating a huge dataset quickly and accurately.

```
1  > mean(Discoveries[,2])
2  [1] 3.1
3
4  > round(mean(Discoveries[,2]))
5  [1] 3
6
7  > max(Discoveries[,2])
8  [1] 12
9
10 #which function helps you get the index of matching condition
11 > which(Discoveries[,2]==12)
12 [1] 26
13
14 > Discoveries[26,]
15    year count
16 26 1885    12
```

To obtain quick summary statistics on a data object, use the summary() function.

```
> summary(Discoveries)
      year          count
 Min.   :1860   Min.   : 0.0
 1st Qu.:1885   1st Qu.: 2.0
 Median :1910   Median : 3.0
 Mean   :1910   Mean   : 3.1
 3rd Qu.:1934   3rd Qu.: 4.0
 Max.   :1959   Max.   :12.0
```

To sum a column in a data frame, use the colSums() function.

```
 > colSums(Discoveries[2])
count
  310
```

Note 1: Note that the colSums() function requires a vector reference rather than a data frame, therefore, no comma.

Note 2: Note the camel casing in the function name. camelCase is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter. Remember R is case-sensitive.

Running queries on data frames

The which() function allows you to specify a query over a data set. Queries are an important tool for data analysis. Queries allow you to identify or locate a particular data record or record set.

```
# how many years were fewer than 5 discoveries observed?
> length(which(Discoveries[,2] < 5))
[1] 79

#  In which years were fewer than 5 discoveries observed?
> Discoveries[(which(Discoveries[,2] < 5)),]
    year count
2   1861     3
3   1862     0
4   1863     2
5   1864     0

# List of years with 0 discoveries
> Discoveries[(which(Discoveries[,2] == 0)),1]
[1] 1862 1864 1881 1904 1917 1933 1956 1957 1959
```

Dealing with missing data

Missing data values in a data frame are encoded as NA. In R, a missing value restricts any calculation of summary statistics or numeric expressions on the data. In R, in order to perform statistical operations on a numeric vector, the missing values need to be removed. Missing values can be removed using the built-in R function na.omit. Some functions accept an argument that allows the caller of the function to specify that the NA values should be removed, the argument is called na.rm. To determine if a dataset has missing values, use the function any() or is.na().

The following code, uses these functions while examining the built in dataset "airquality"; "airquality" contains measurements of daily air quality in New York City from May through September 1973.

```
1  > head(airquality)
2  Ozone Solar.RWind Temp Month Day
3  1 41 190 7.4 67 5 1
4  2 36 118 8.0 72 5 2
5  3 12 149 12.6 74 5 3
6  4 18 313 11.5 62 5 4
7  5 NANA14.3 56 5 5
8  6 28 NA14.9 66 5 6
9
10 > mean(airquality$Solar.R)
11 [1] NA
12 Warning message:
13 In mean.default(airquality) :
14   argument is not numeric or logical: returning NA
15
16 > any(is.na(airquality))
17 [1] TRUE
18
19 > mean(airquality$Solar.R,na.rm=TRUE)
20 [1] 185.9315
21
22 > which(is.na(airquality$Solar.R))
23 [1] 5 6 11 27 96 97 98
24
25 > air_complete<-na.omit(airquality)
26 > head(air_complete)
27 Ozone Solar.RWind Temp Month Day
28 1 41 190 7.4 67 5 1
29 2 36 118 8.0 72 5 2
30 3 12 149 12.6 74 5 3
```

```
31  4 18 313 11.5 62 5 4
32  7 23 299 8.6 65 5 7
33  8 19 99 13.8 59 5 8
```

Saving R scripts

We recommend using RStudio to create your R scripts. However, R commands can be written in a text file and loaded on demand. Create a text file in a text editor and save the file with the .R extension. Use the source() function to load and execute the script.

```
1   # Simple R script: created.R
2   seq1<-1:10
3   seq2<-seq(from=100,to=300,length=10)
4   df<-data.frame(seq1,seq2)
5
6   > source("created.R")
7   > df
8       x        y
9   1   1 100.0000
10  2   2 122.2222
11  3   3 144.4444
12  4   4 166.6667
13  5   5 188.8889
14  6   6 211.1111
15  7   7 233.3333
16  8   8 255.5556
17  9   9 277.7778
18  10 10 300.0000
```

The source function is useful when you are dealing with a large dataset and loading the data set takes time.

Saving .Rdata file

Rstudio allows you to save the current workspace as a .Rdata file. When starting Rstudio after a system shutdown you can load the .Rdata workspace; this will cause all the objects loaded previously to be restored. Basically this functionality comes in handy when we are dealing with big data and loading of files in R takes a lot of time. In these scenarios, it is quicker to save the workspace so that you do not have to load the data files again.

Conditional Statements

The if() statement is used to construct conditional execution paths. In conditional execution, code statements are only executed if certain conditions are TRUE. The conditional code statements are enclosed in curly braces { and }.

```
1  > num1<-10
2  > num2<-5
3  > if(num1 > num2) {
4  + print("a is less than b")
5  + }
6
7  [1] "a is less than b"
```

Logical operators

Operators	Semantics
==	Equality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Binary Operators

Operators	Semantics
&&	AND (both statements are true)
||	OR (Atleast one statement is true
!	NOT (Statement is false)

Nested IF statements

```
1  > if (sum(1:10) >= sqrt(75)) {
2  + print("true")
3  + } else {
4  + print("false")
5  + }
6  [1] "true"
```

The ifelse() function provides a more compact syntax for if-else constructs.

```
1  > ifelse(sum(1:5) >= 10, "it's greater", "it's smaller")
2  [1] "it's greater"
```

Switch statements

Switch statements are used when a particular variable can have multiple cases and and different execution code is associated with the different values.

```
1  #readline function can be used to get a user input
2  > name <- readline(prompt="Enter a name: ")
3  Enter a name: Michelle
4
5  > switch(name,
6  + Michelle={
7  + print("Hi Michelle! How are you?") # any logical statement for Michelle
8  + },
9  + John={
10 + print("Hi John! How are you?") # any logical statement for John
11 + },
12 + {
13 + Print("default") #default logic
14 + }
15 + )
16
17 [1] "Hi Michelle! How are you?"
```

Control structures

R supports two common forms of iteration (looping):

- restricted iteration which executes commands a fixed number of times: for loop
- unrestricted iteration in which the loop runs until some condition is no longer true: while loop

The for loop runs a fixed number of times based on the values assigned to an index or looping variable.

```
1  > for (i in 1:3) {
2  + print(paste("i =",i))
3  + }
4
5  [1] "i = 1"
6  [1] "i = 2"
7  [1] "i = 3"
8
9  > i
10 [1] 3
```

Instead of looping a fixed number of times, a for loop can also iterate over a set. The loop variable takes on each value in the set one at a time.

```
1  > cities <-c("Boston","NewYork","SanFrancisco")
2  > for (city in cities) {
3  + print(city)
4  + }
5
6  [1] "Boston"
7  [1] "New York"
8  [1] "San Francisco"
```

For loops can be nested to run through each row and column of a matrix.

```
1  > mat<- matrix(nrow=4, ncol=5, sample(0:1))
2  > mat
3       [,1] [,2] [,3] [,4] [,5]
4  [1,]    0    0    0    0    0
5  [2,]    1    1    1    1    1
6  [3,]    0    0    0    0    0
7  [4,]    1    1    1    1    1
8
9  > for (i in 1:nrow(mat) ) {
10 + for (j in 1:ncol(mat)){
11 + if(mat[i,j] == 1){
12 + mat[i,j]<- "Michelle"
13 + }
14 + else{
15 + mat[i,j]<- "John"
16 + }
17 + }
```

```
18  + }
19
20  > mat
21       [,1]       [,2]       [,3]       [,4]       [,5]
22  [1,] "John"     "John"     "John"     "John"     "John"
23  [2,] "Michelle" "Michelle" "Michelle" "Michelle" "Michelle"
24  [3,] "John"     "John"     "John"     "John"     "John"
25  [4,] "Michelle" "Michelle" "Michelle" "Michelle" "Michelle"
```

In an unrestricted iteration, the loop executes the loop statements until a condition is no longer true.

```
 1  > x <-0
 2  > while (x < 10) {
 3  + print (x)
 4  + x <-x + 1
 5  + }
 6
 7  [1] 0
 8  [1] 1
 9  [1] 2
10  [1] 3
11  [1] 4
12  [1] 5
13  [1] 6
14  [1] 7
15  [1] 8
16  [1] 9
```

Apply function

The apply function allows you to apply a function to a list of values. It is a form of functional programming. Think of it as an alternative method of looping thriough the values of a list and applying the same code to each value in the list.

> apply(): Applies a function to components of a list or other object, and then returns the results as a list, a vector, or a matrix.

```
1  > x <- matrix(c(1:10), ncol=5, byrow=TRUE)
2  > x
3       [,1] [,2] [,3] [,4] [,5]
4  [1,]    1    2    3    4    5
5  [2,]    6    7    8    9   10
6  > apply(x, 1, mean)
7  [1] 3 8
8
9  > apply(x, 2, mean)
10 [1] 3.5 4.5 5.5 6.5 7.5
```

> Note: apply(x, 1, mean) calculates the mean of two rows in x, and apply(x, 2, mean) calculates the mean of five columns in x.

In the above example, apply extracts each column/row as a vector, one at a time and passes the vector to the mean function. It substitutes the use of loop. An alternative solution to the above code can be written as:

```
1  > avgs<-numeric(5)
2  > for(i in 1:5){
3  +   avgs[i]<-mean(x[,i])
4  + }
5
6  > avgs
7  [1] 3.5 4.5 5.5 6.5 7.5
8
9  #OR
10 > apply(x, 2, mean)
11 [1] 3.5 4.5 5.5 6.5 7.5
```

The looping mechanism in R, is relatively slow when compared to apply, this is especially true for large datasets. The apply function reduces the processing time considerably, since the looping mechanism in the apply function is done in compiled code, like c or Fortran, not in R's own interpreted code.

Types of apply

- lapply - L in lapply stands for list. So lapply(x) returns a list of the same length of x

```
1  > x<- list(a<-c(1:20),b<-c(10:20),c<-c(20:30))
2  > x
3  [[1]]
4   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
5  
6  [[2]]
7   [1] 10 11 12 13 14 15 16 17 18 19 20
8  
9  [[3]]
10  [1] 20 21 22 23 24 25 26 27 28 29 30
11 
12 > results<-lapply(x,mean)
13 > results
14 [[1]]
15 [1] 10.5
16 
17 [[2]]
18 [1] 15
19 
20 [[3]]
21 [1] 25
22 
23 > class(results)
24 [1] "list"
```

- sapply - S stands for simplifying. Sapply works like lapply but instead of returning a list it returns a simple vector. It accepts an argument simplify. If simplify=TRUE, then it returns a simple vector. If simplify=FALSE, it behaves like lappy. The defaulT value for simplify is TRUE.

```
1  > results<-sapply(x,mean)
2  > results
3  [1] 10.5 15.0 25.0
4  
5  > class(results)
6  [1] "numeric"
```

- tapply - It is used to apply a function to subsets of a vector and the subsets are defined by some other vector, usually a factor. It is typically a categorical variable that is used to group the value of the vector into bins. The function is applied to each group defined by the categorical variable.

```
1  > x <- 1:20
2  > x
3   [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
4
5  > y <- factor(rep(letters[1:5], each = 4))
6  > y
7   [1] a a a a b b b b c c c c d d d d e e e e
8  Levels: a b c d e
9
10 > tapply(x, y, mean)
11    a    b    c    d    e
12  2.5  6.5 10.5 14.5 18.5
```

- mapply - mapply is used when you want to apply a function to the 1st element of each and then the 2nd elements of each etc. In line 1 below, we apply sum to 3 rows that consist of 5 columns who values are all 1's in the first column to all 5's in the last column. We have 5 values in the result vector since we are summing the values for each column.

```
1  > mapply(sum, 1:5, 1:5, 1:5)
2  [1]  3  6  9 12 15
3
4  > mapply(rep, 1:4, 4:1)
5  [[1]]
6  [1] 1 1 1 1
7  [[2]]
8  [1] 2 2 2
9  [[3]]
10 [1] 3 3
11 [[4]]
12 [1] 4
```

Split-Apply-Combine strategy

The methodology for functional programming involves:

- Breaking big problems into small, manageable chunks of code
- Performing operations on each chunk of code separately.
- Combining the output of each piece into a single output.

The "Plyr" package provides intuitive functions for split-apply-combine strategy

Input	Output			
	Array	Data frame	List	Discarded
Array	aaply	adply	alply	a_ply
Data frame	daply	ddply	dlply	d_ply
List	laply	ldply	llply	l_ply

Fig. 3a - plyr package

Example:

1. Split the iris dataset into three parts.
2. Remove the species name variable from the data.
3. Calculate the mean of each variable for the three different parts separately.
4. Combine the output into a single data frame.

```
1  > library (plyr)
2  # Here tilde operator is taking each species value at every iteration.
3
4  > ddply(iris,~Species,function(x) colMeans(x[,-which(colnames(x)=="Species")]))
5        Species Sepal.Length Sepal.Width Petal.Length Petal.Width
6
7  1      setosa        5.006       3.428        1.462       0.246
8  2 versicolor        5.936       2.770        4.260       1.326
9  3  virginica        6.588       2.974        5.552       2.026
10
11 #OR
12
13 > iris_mean <- adply(iris3,3,colMeans)
14 > iris_mean
15           X1 Sepal L. Sepal W. Petal L. Petal W.
16 1      Setosa    5.006    3.428    1.462    0.246
17 2 Versicolor    5.936    2.770    4.260    1.326
18 3  Virginica    6.588    2.974    5.552    2.026
19
20 > class(iris_mean)
21 [1] "data.frame"
```

Note: You need to install the plyr package to run this code.

Reading and writing data

The ability to read and write external text files is an essential part of data processing. Many data sets are stored in simple text files. Excel and other programs can export and import text files in certain formats.

Lets load the built-in data set AirPassengers containing monthly international airline passenger data between 1949 and 1960. After displaying the data set, copy the data into a simple text file.

```
1  > AirPassengers
2       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
3  1949 112 118 132 129 121 135 148 148 136 119 104 118
4  1950 115 126 141 135 125 149 170 170 158 133 114 140
5  1951 145 150 178 163 172 178 199 199 184 162 146 166
6  1952 171 180 193 181 183 218 230 242 209 191 172 194
7  1953 196 196 236 235 229 243 264 272 237 211 180 201
8  1954 204 188 235 227 234 264 302 293 259 229 203 229
9  1955 242 233 267 269 270 315 364 347 312 274 237 278
10 1956 284 277 317 313 318 374 413 405 355 306 271 306
11 1957 315 301 356 348 355 422 465 467 404 347 305 336
12 1958 340 318 362 348 363 435 491 505 404 359 310 337
13 1959 360 342 406 396 420 472 548 559 463 407 362 405
14 1960 417 391 419 461 472 535 622 606 508 461 390 432
```

While R has several functions for reading files, the most commonly used function for reading text files is read.table(). When calling read.table, the first argument contains the name of the file you wish to open. You can also specify the header argument and the separator argument. The header argument specifies if the data has a row with variable names. The separator character isthe delimiting character, or the delimiter, that separate the data values from each other. In the example below a space separates the data values from each other.

```
1  > ap<-read.table("airPassengers.txt",header=TRUE,sep="")
2  > ap
3       Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
4  1949 112 118 132 129 121 135 148 148 136 119 104 118
5  1950 115 126 141 135 125 149 170 170 158 133 114 140
6  1951 145 150 178 163 172 178 199 199 184 162 146 166
7  1952 171 180 193 181 183 218 230 242 209 191 172 194
8  1953 196 196 236 235 229 243 264 272 237 211 180 201
9  1954 204 188 235 227 234 264 302 293 259 229 203 229
10 1955 242 233 267 269 270 315 364 347 312 274 237 278
11 1956 284 277 317 313 318 374 413 405 355 306 271 306
12 1957 315 301 356 348 355 422 465 467 404 347 305 336
```

```
13  1958 340 318 362 348 363 435 491 505 404 359 310 337
14  1959 360 342 406 396 420 472 548 559 463 407 362 405
15  1960 417 391 419 461 472 535 622 606 508 461 390 432
```

The read.table() function has a skip=x parameter which allows you to skip some number of lines.

```
1  > ap<-read.table("airPassengers.txt",skip=4,header=TRUE,sep="")
2  > ap
3    X1952 X171 X180 X193 X181 X183 X218 X230 X242 X209 X191 X172 X194
4  1  1953  196  196  236  235  229  243  264  272  237  211  180  201
5  2  1954  204  188  235  227  234  264  302  293  259  229  203  229
6  3  1955  242  233  267  269  270  315  364  347  312  274  237  278
7  4  1956  284  277  317  313  318  374  413  405  355  306  271  306
8  5  1957  315  301  356  348  355  422  465  467  404  347  305  336
9  6  1958  340  318  362  348  363  435  491  505  404  359  310  337
10 7  1959  360  342  406  396  420  472  548  559  463  407  362  405
11 8  1960  417  391  419  461  472  535  622  606  508  461  390  432
```

NOTE - read.table function converts the data into factors. when you do not desire this conversion, use stringsAsFactors=FALSE as an argument.

New columns can be added to a data set using the cbind() function.

```
1  > ap$Total<-cbind(rowSums(ap))
2  > ap
3    X1952 X171 X180 X193 X181 X183 X218 X230 X242 X209 X191 X172 X194 Total
4  1  1953  196  196  236  235  229  243  264  272  237  211  180  201  4653
5  2  1954  204  188  235  227  234  264  302  293  259  229  203  229  4821
6  3  1955  242  233  267  269  270  315  364  347  312  274  237  278  5363
7  4  1956  284  277  317  313  318  374  413  405  355  306  271  306  5895
8  5  1957  315  301  356  348  355  422  465  467  404  347  305  336  6378
9  6  1958  340  318  362  348  363  435  491  505  404  359  310  337  6530
10 7  1959  360  342  406  396  420  472  548  559  463  407  362  405  7099
11 8  1960  417  391  419  461  472  535  622  606  508  461  390  432  7674
```

A data object can be exported to a file using the write.table() function.

```
1  > getwd()
2  [1] "C:/Users/Martin/Downloads"
3
4  > write.table(ap,"AirPassNG.txt",col.names=NA,
5  row.names=TRUE,quote=FALSE,sep=",")
```

Note: R requires the use of a forward slash (‘/’) to separate directories (folders) not the backslash (‘\’) used by Windows.

Chapter 4

Data shaping

Date processing

Dates are an important data format and require special processing. R has built-in functions to deal with date values encoded in different formats. The built-in as.date() function can only handle dates but not time values. Better support for date processing is provided by the *lubridate* package.

The base object date stores date internally as a number of days elapsed since January 1, 1970.

```
1  > as.Date("1970-01-01")
2  [1] "1970-01-01"
3  > as.numeric(as.Date("1970-01-02"))
4  [1] 1
5  > as.numeric(as.Date("1970-01-01"))
6  [1] 0
```

By default, the date object requires a string is the format "YYYY-MM-DD", but other format specification are possible:

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a %A	abbreviated weekday unabbreviated weekday	Mon Monday
%m	month (00-12)	00-12
%b %B	abbreviated month unabbreviated month	Jan January
%y %Y	2-digit year 4-digit year	07 2007

Fig. 4a - Date Formats

Use ?strptime to get a full description of date formatting strings.

```
1  > d<-as.Date("Dec-27-2014",format="%b-%d-%Y")
2  > as.numeric(d)
3  [1] 16431
4  > dt1 <-as.Date("12/29/2014", format="%m/%d/%Y")
5  > dt1
6  [1] "2014-12-29"
7  > dt2 <-as.Date("Oct-11-14", format="%b-%d-%y")
8  > dt2
9  [1] "2014-10-11"
10 > dt3 <-as.Date("17.12.14", format="%d.%m.%y")
11 > dt3
12 [1] "2014-12-17"
13 > dt4 <-as.Date("December17, 2014", format="%B %d, %Y")
14 > dt4
15 [1] "2014-12-17"
```

> *Note* - The format argument above is not the format of your output, it is the format of your input date which helps the as.date function to understand any kind of input date given the format argument.

R supports calculations on dates, including differences between dates, adding and subtracting days and dates. Remember to convert strings to dates using the as.Date function. In the example below the dates array in line 1 contains character strings NOT dates. Line 4 generates an error since you are performing the subtraction operation on strings nor dates.

```
1  > dates<- c("12/29/2014","Oct-11-14", "December17, 2014")
2  > is.character(dates)
3  [1] TRUE
4  > dates[1] - dates[2]
5  Error in dates[1] - dates[2] : non-numeric argument to binary operator
6  > date1<- as.Date(dates[1], format="%m/%d/%Y")
7  > date2<- as.Date(dates[2], format="%b-%d-%y")
8  > date1-date2
9  Time difference of 79 days
```

R supports vectors (collections) of dates and can calculate the interval between them. Again, functions that perform date data calculations require that the dates have the same format, dates with the different format will return NA.

```
1 > dts<-as.Date(c("2014-06-01","2014-07-08","2014-10-14","Oct-11-14"))
2 > diff(dts)
3 Time differences in days
4 [1] 37 98 NA
```

Time processing

In addition to dates, time variables are also an important data value in data science. Time values are intrinsically processed using the POSIXctclass. The POSIXctclass represents combined date and time value.

```
1 > tm1 <-as.POSIXct("2014-12-28 09:59:43")
2 > tm1
3 [1] "2014-12-28 09:59:43 EST"
4 > class(tm1)
5 [1] "POSIXct" "POSIXt"
```

Code	Meaning	Code	Meaning
%a	Abbreviated weekday	%A	Full weekday
%b	Abbreviated month	%B	Full month
%c	Locale-specific date and time	%d	Decimal date
%H	Decimal hours (24 hour)	%I	Decimal hours (12 hour)
%j	Decimal day of the year	%m	Decimal month
%M	Decimal minute	%p	Locale-specific AM/PM
%S	Decimal second	%U	Decimal week of the year (starting on Sunday)
%w	Decimal Weekday (0=Sunday)	%W	Decimal week of the year (starting on Monday)
%x	Locale-specific Date	%X	Locale-specific Time
%y	2-digit year	%Y	4-digit year
%z	Offset from GMT	%Z	Time zone (character)

Fig. 4b - Different formats for date and time

```
1 > tm2 <-as.POSIXct("2014-12-19 11:38:42", tz="GMT")
2 > tm2
3 [1] "2014-12-19 11:38:42 GMT"
4 > tm3 <-as.POSIXct("25072013 08:32:07", format = "%d%m%Y %H:%M:%S")
5 > tm3
6 [1] "2013-07-25 08:32:07 EDT"
```

Like date calculations, in order to perform time operations on two different time variables, the format should be same.

```
1  > tm1 <-as.POSIXct("2013-07-24 23:55:26")
2  > tm1
3  [1] "2013-07-24 23:55:26 EDT"
4  > tm2 <-as.POSIXct("25072013 08:32:07", format = "%d%m%Y%H:%M:%S")
5  > tm2
6  [1] "2013-07-25 08:32:07 EDT"
7  > tm2 -tm1
8  Time difference of 8.611389 hours
9  > tm1 + 25
10 [1] "2013-07-24 23:55:51 EDT"
```

To get the current time and date as a POSIXct object, use the Sys.time()functions.

```
1  > Sys.time()
2  [1] "2015-07-08 17:46:33 EDT"
```

The lubridate package offers the same functionality as the function now(). Lubricate also provides the today() function; it returns the current date.

```
1  > library("lubridate")
2  > now()
3  [1] "2015-07-08 18:02:29 EDT"
4
5  > today()
6  [1] "2015-07-08"
7
8  > class(today())
9  [1] "Date"
```

The POSIXltclass enables easy extraction of specific components of a time variable and makes the calculation of specific components easier.

```
1  > tm3 <-as.POSIXlt("2015-03-03 05:35:10")
2  > tm3$hour
3  [1] 5
4  > tm3$min
5  [1] 35
6  > tm3$hour<- tm3$hour+2
7  > tm3
8  [1] "2015-03-03 07:35:10 EST"
```

While the built-in time and date processing of R is often sufficient, there are several packages that simplify certain date and time processing tasks. The most popular package for date processing is lubridate. Other packages like chron, timewarp, date are also available for date processing and manipulation, each offering some advantage over the other.

Packages

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored is called the library. R comes with a standard set of packages, but others are available for download. Once installed, they have to be loaded into the session before they can be used.

Important functions used with packages:

- .libPaths() - to get library location
- library() - to list all installed packages
- search() - to list currently loaded packages
- require() - to load a package for use

Installing a package in R

Steps to install a package in R:

1.Select Packages/Install package(s)…

2.Select the closest download site

3.Choose the package let's say 'lubridate' from the list

4.Select Packages/Install package(s) from local zip files…

5.Select the directory into which the package was downloaded

6.Select the package zip file

The package is now installed and needs to be loaded using the library(lubridate)function

The same process can be done in one step in R studio by running the following command.

```
1 install.packages("lubridate")
```

Lubridate package

The lubridate package is one of several date processing add-on packages available for R. It provides more intuitive handling of date and time values. The library is a wrapper around the POSIXctclass with more intuitive syntax. Using lubridate requires installation of the package first.

The lubridate package has numerous functions for creating date and time objects that do not require a parse format string.

Order of elements in date-time	Parse function
year, month, day	ymd()
year, day, month	ydm()
month, day, year	mdy()
day, month, year	dmy()
hour, minute	hm()
hour, minute, second	hms()
year, month, day, hour, minute, second	ymd_hms()

Fig. 4c - lubridate functions

```
1  > tm1.lub <-ymd_hms("2014-12-28 10:26:23")
2  > tm1.lub
3  [1] "2014-12-28 10:26:23 UTC"
4  >
5  > tm2.lub <-mdy_hm("12/28/14 11:44")
6  > tm2.lub
7  [1] "2014-12-28 11:44:00 UTC"
8  > tm3.lub <-dmy_hm("28.12.14 9:30AM")
9  > tm3.lub
10 [1] "2014-12-28 09:30:00 UTC"
```

The lubridate package simplifies the extraction of date and time components.

Date component	Accessor
Year	year()
Month	month()
Week	week()
Day of year	yday()
Day of month	mday()
Day of week	wday()
Hour	hour()
Minute	minute()
Second	second()
Time zone	tz()

Fig. 4d - Accessor functions of lubridate

```
1  > year(tm3.lub)
2  [1] 2014
3
4  > wday(tm3.lub, label=TRUE)
5  [1] Sun
6  Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat
7
8  > hour(tm3.lub)
9  [1] 9
10 > tz(tm3.lub)
11 [1] "UTC"
```

The lubridate package also simplifies updating date and time components of a POSIXctobject. The update() function allows the parts of a date (such as day, month, year) to be updated in a POSIXctobject.

```
1  > tm3.lub <-dmy_hm("28.12.14 9:30AM")
2  > tm3.lub
3  [1] "2014-12-28 09:30:00 UTC"
4
5  > year(tm3.lub) <-2013
6  > tm3.lub
7  [1] "2013-12-28 09:30:00 UTC"
8
9  > update(tm3.lub, year = 2010, month = 1, day = 1)
10 [1] "2010-01-01 09:30:00 UTC"
```

To facilitate time calculations, convert a time to a decimal value.

```
1  > tm3.lub
2  [1] "2013-12-28 09:30:00 UTC"
3
4  > tm3.dechr <-hour(tm3.lub) + minute(tm3.lub)/60 + second(tm3.lub)/3600
5  > tm3.dechr
6  [1] 9.5
```

Suppose that within a dataset, instead of a reference to a full date, only the month of the date is available. To address this, we need to add a dummy month and year to convert this string to a proper date format. Once the format is in a date format, we can easily convert the string to a numeric month or vice versa.

```
> mon<-"Sep"
> date<-"2015-Sep-01"
> mydate<- as.Date(date,format="%Y-%b-%d")
> mydate
[1] "2015-09-01"

> format(mydate,"%d")
[1] "01"
> format(mydate,"%m")
[1] "09"
```

Text Processing

R has many built-in functions to process text. Some common text functions are: paste(), nchar(), substr() etc.

```
> pdf<- "stringr.pdf"
#paste command with separator argument set to no space
> paste("http://cran.r-project.org/web/packages/stringr/",pdf,sep="")
[1] "http://cran.r-project.org/web/packages/stringr/stringr.pdf"

# no sep argument and thus there is a space
> paste("http://cran.r-project.org/web/packages/stringr/",pdf)
[1] "http://cran.r-project.org/web/packages/stringr/ stringr.pdf"

> nchar(pdf)
[1] 11

> substr(pdf,3,6)
[1] "ring"
```

stringr package

The stringr package has many functions that simplify text manipulation.

Base R Function	*stringr* Function
`paste()` – concatenates a vector of characters with spaces in between	`str_c()` – similar to paste() but does not insert extra spaces and removes empty strings
`nchar()` – returns the length of a string. For NA it returns 2.	`str_length()` – same as `nchar()` but handles NA correctly.
`substr()` – extracts or replaces a substring sequence	`str_sub()` – similar to `substr()` but also accepts negative values for offsets.
	`str_dup()` – duplicates a string
	`str_trim()` – removes leading and trailing spaces
	`str_pad()` – pad string with extra whitespace on left or right

Fig. 4e - Text processing functions

```
library("stringr")
> str_c("http://cran.r-project.org/web/packages/stringr/",pdf)

[1] "http://cran.r-project.org/web/packages/stringr/stringr.pdf"

> fac<-factor(c(0,1,1,0,1,0,0,1), labels=c("False","True"))
> fac
[1] False True  True  False True  False False True
Levels: False True

#str_length can handle factors whereas nchar cannot
> str_length(fac)
[1] 5 4 4 5 4 5 5 4
> nchar(fac)
Error in nchar(fac) : 'nchar()' requires a character vector

#str_sub can handle factors whereas substr cannot
> str_sub(fac,1,3) #arguments vector, Start, End
[1] "Fal" "Tru" "Tru" "Fal" "Tru" "Fal" "Fal" "Tru"

> substr(fac,-4,-2)
```

```
22  [1] "" "" "" "" "" "" "" ""
```

The str_split function helps split the string into two strings just like a subset.

```
1  > library(stringr)
2  > str<-"01-04-2013T12:32:43"
3
4  > split<-str_split(str,"T")
5  > split
6  [[1]]
7  [1] "01-04-2013" "12:32:43"
```

The str_trim function removes leading and trailing spaces from a text string. We would suggest to always use this function when dealing with text.

```
1  > str<-"    Leading spaces and trailing spaces    "
2  > str_trim(str)
3  [1] "Leading spaces and trailing spaces"
```

All the other stringr package functions are very similar to some other built-in functions of R, for example, str_detect and str_extract are very similar to the grep function.

Regular Expression

A regular expression (regex) is a special text string that specifies a search pattern. Similar to wild cards, Searching all files in the directory: *.txt, Regular expression: ".txt$"*. Also known as Reg Exp or regex.

Regex in R can be somewhat different from regex in any other languages, but the essential basics remain the same, for example metacharacters used in R are the same as regex other languages.

Metacharacters

Metacharacters are special characters which have specific meaning of their own. Most characters, including letters and digits, when used in a regex will match themselves, but metacharacters are different, for example the Pattern 'Plus+' will not match the pattern 'Plus+'.

- '.' Normally matches any one character except a newline. Within square brackets, the dot is literal.
- '[]' Groups a series of pattern elements to a single element.
- '+' Matches preceding operator one or more times.
- '?' Matches preceding operator zero or one time

Examples:

- [hc]at: matches "hat" or "cat"
- .at: matches any 3 characters ending with "at"

```
1  #str_replace is another stringr function which takes three
2  # argument string, match and replacement
3  > text<- c("bat","hat","match","mismatch")
4  > str_replace(text, ".at","*123*")
5  [1] "*123*"      "*123*"      "*123*ch"    "mis*123*ch"
6
7  #replacing at or a i.e. replacing if t is there (1) or if t is not there (0)
8  > text<- c("bat","hat","match","mismatch","man")
9  > str_replace(text, "at?","*123*")
10 [1] "b*123*"      "h*123*"      "m*123*ch"    "mism*123*ch" "m*123*n"
11
12 > text<- c("bat","hat","match","mismatch")
13 > str_replace(text, "a+","*123*")
14 [1] "b*123*t"      "h*123*t"      "m*123*tch"    "mism*123*tch"
```

Escaping metacharacters

Metacharacters are a very powerful tool to implement regex successfully, but the problem arises when we want to match a metacharacter as a character. What if there is a need to search 'Plus+' literally? Escaping metacharacters is a mechanism to use when you want to match a literal metacharacter expression as oppose to interpret a metacharacter expression.

Metacharacters and how to escape them in R

Metacharacter	Literal meaning	Escape in R
.	the period or dot	\\.
$	the dollar sign	\\$
*	the asterisk or star	*
+	the plus sign	\\+
?	the question mark	\\?
|	the vertical bar or pipe symbol	\\\|
\	the backslash	\\\\
^	the caret	\\^
[	the opening square bracket	\\[
]	the closing square bracket	\\]
{	the opening curly bracket	\\{
}	the closing curly bracket	\\}
(	the opening round bracket	\\(
)	the closing round bracket	\\)

Fig. 4f - Escaping metacharacters

```
1 > per<- "plus+"
2 > str_replace(per, "plus+", "replaced")
3 [1] "replaced+"
4 > str_replace(per, "plus\\+", "replaced")
5 [1] "replaced"
```

R supports two different forms of regular expressions.

- Extended regular expression: They use an implementation of the POSIX 1003.2 standard: that allows some scope for interpretation and the interpretations here are those currently used by R.

POSIX Character Classes in R

Class	Description
[[:lower:]]	Lower-case letters
[[:upper:]]	Upper-case letters
[[:alpha:]]	Alphabetic characters ([[:lower:]] and [[:upper:]])
[[:digit:]]	Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
[[:alnum:]]	Alphanumeric characters ([[:alpha:]] and [[:digit:]])
[[:blank:]]	Blank characters: space and tab
[[:cntrl:]]	Control characters
[[:punct:]]	Punctuation characters: ! " # % & ' ( ) * + , - . / : ;
[[:space:]]	Space characters: tab, newline, vertical tab, form feed, carriage return, and space
[[:xdigit:]]	Hexadecimal digits: 0-9 A B C D E F a b c d e f
[[:print:]]	Printable characters ([[:alpha:]], [[:punct:]] and space)
[[:graph:]]	Graphical characters ([[:alpha:]] and [[:punct:]])

Fig. 4g - Extended regex

- Perl-like regular expression: pattern matching using the same syntax and semantics as Perl 5.10

Anchor Sequences in R

Anchor	Description
\\d	match a digit character
\\D	match a non-digit character
\\s	match a space character
\\S	match a non-space character
\\w	match a word character
\\W	match a non-word character
\\b	match a word boundary
\\B	match a non-(word boundary)
\\h	match a horizontal space
\\H	match a non-horizontal space
\\v	match a vertical space
\\V	match a non-vertical space

Fig. 4h - Perl like regex

Example:

```
1  > library(stringr)
2  > email<-"yatish.jain@gmail.com"
3
4  > str_match(email,"\\@\\w+\\.") #getting both @ and .
5
6       [,1]
7  [1,] "@gmail."
8
9  > str_match(email,"\\@(\\w+)\\.") #adding parenthesis to desired text
10
11      [,1]      [,2]
12 [1,] "@gmail." "gmail"
13
14 > str_match(email,"\\@(\\w+)\\.")[[2]]
15
16 [1] "gmail"
```

Grep function

The grep function is a very powerful and very important function for text processing. It searches for matches to an argument pattern within each element of a character vector. Grep takes two value arguments:

- If you pass value=FALSE, returns a new vector with the indexes of the elements in the input vector that could be matched by the regular expression.
- If you pass value=TRUE, returns a vector with copies of the actual elements in the input vector that could be matched.

```
1  > text<- c("bat","hat","match","mismatch")
2  > grep("ma?",text, value=T)
3  [1] "match"    "mismatch"
4
5  > grep("ma?",text, value=F)
6  [1] 3 4
```

Grep function can be used to test the regex on a small subset of data before implementing the regex on big data for cleaning or extracting data, this way you can check the result of your regex quickly.

```
1  > text<- c("bat","hat","//match?","?mismatch")
2  > grep("\\//",text, value=F)
3  [1] 3
4
5  > text1<-str_replace(text,"\\//", "")
6  > str_replace(text1,"\\?", "")
7  [1] "bat"      "hat"      "match"    "mismatch"
```

Chapter 5

Algorithmic Complexity

Complexity is a way to approximate the time and space required for an algorithm or program to execute. Algorithmic complexity quantifies how fast or slow an algorithm is. Complexities are used to compare algorithms on a conceptual level, i.e., ignoring low-level details.

Algorithmic complexity can be defined as the time taken by any algorithm without depending on its implementation details, but if you think about it, a given algorithm will take different amounts of time on the same input depending on factors such as: computer load, compiler used, processor speed, disk speed, instruction set etc. The only way around this problem is to consider the efficiency of each algorithm asymptotically i.e., the value it will eventually reach. Thus, we measure time T(n) as the number of steps, given that each step takes constant time.

Let's consider an example. Consider adding two binary integers digit by digit (or bit by bit). Adding a single bit is a "step" in the computation, adding two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c * n, where c is time taken by the addition of two bits and n is the number of times we need to add two bits. Repeating the same process will take different times on two different machines but time T(n) grows linearly as input size increases.

In the above example, the actual execution time is difficult to estimate as it depends on a multitude of factors:

- Type of CPU
- Operating system
- Programming language
- Other processes running concurrently
- Data structures and data representation

Hence instead of worrying about the actual execution time, programmers compare and classify algorithms through their asymptotic growth as a function of the size of the input.

There is often a significant difference between the best, worst, and average runtime of a program. Average class behavior is considered as an acceptable algorithm.

Example: Suppose you need to search a column in a data frame or a vector for a specific element, e.g., find the data for flight "JB721-010214". How many string comparisons would you need to perform if there were n elements?

- On average: n/2
- Worst case: n
- Best case: 1

Asymptotic growth function

In the previous example, the average and the worst case runtime would both double if the input size were doubled. The actual runtime would need to be measured on a specific platform, but we can characterize the runtime behavior to increase linearly with the input size.

The aim of Algorithmic complexity is to classify algorithms according to their performance. Computer Scientists use the big-O notation to capture the essence of the worst case behavior of an algorithm or program. It is a function that describes the growth behavior of runtime or memory/storage requirements for a program. For the previous example, the program would have a time complexity of O(n).

Common complexities

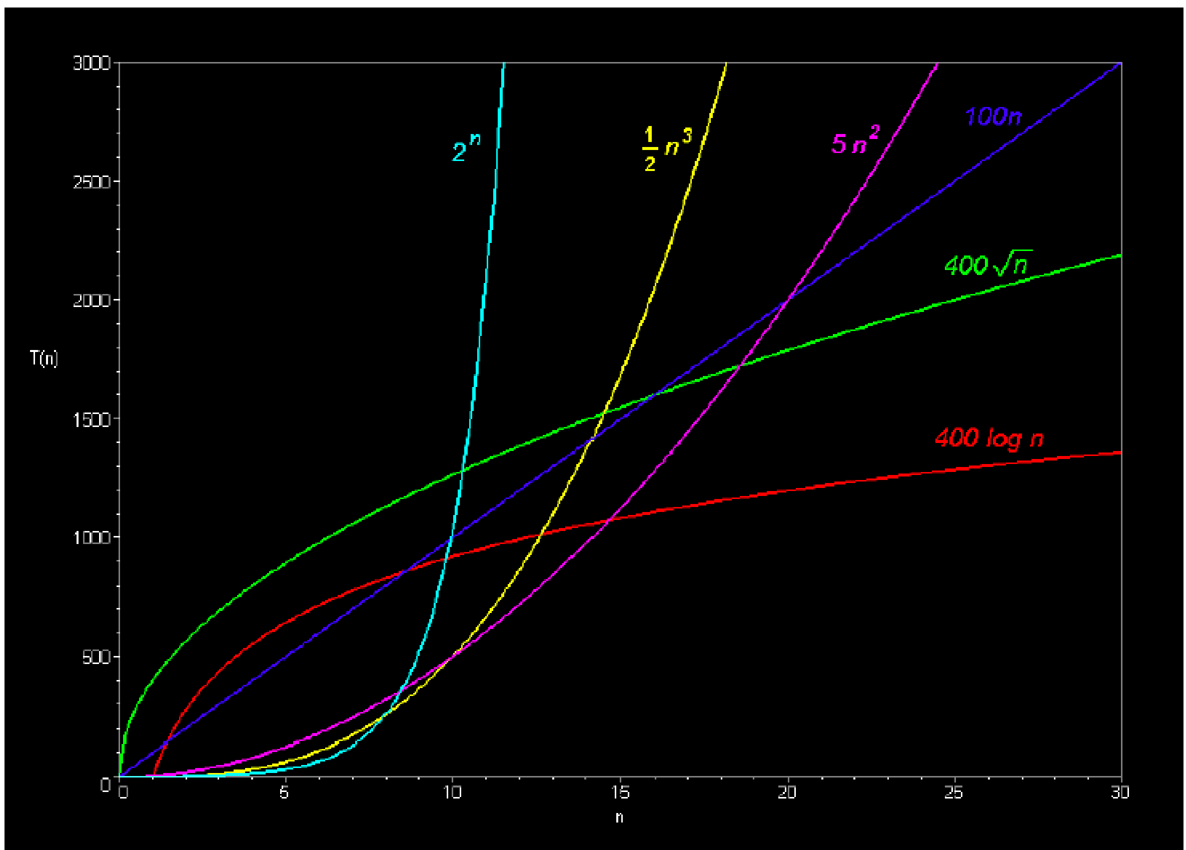


Fig. 5a - Complexities

- Constant - O(1)
- Linear - O(n)
- Quadratic - $O(n^2)$
- Cubic - $O(n^3)$
- Exponential - $O(2^n)$

- Logarithmic - O(log n)
- Log-linear - O(n log n)

Constant

An algorithm is said to be constant time complexity when it requires the same amount of time regardless of input size.

Example - Accessing array element

Linear

An Algorithm runs in linear time when execution time is directly proportional to the input size. Example - Linear array search

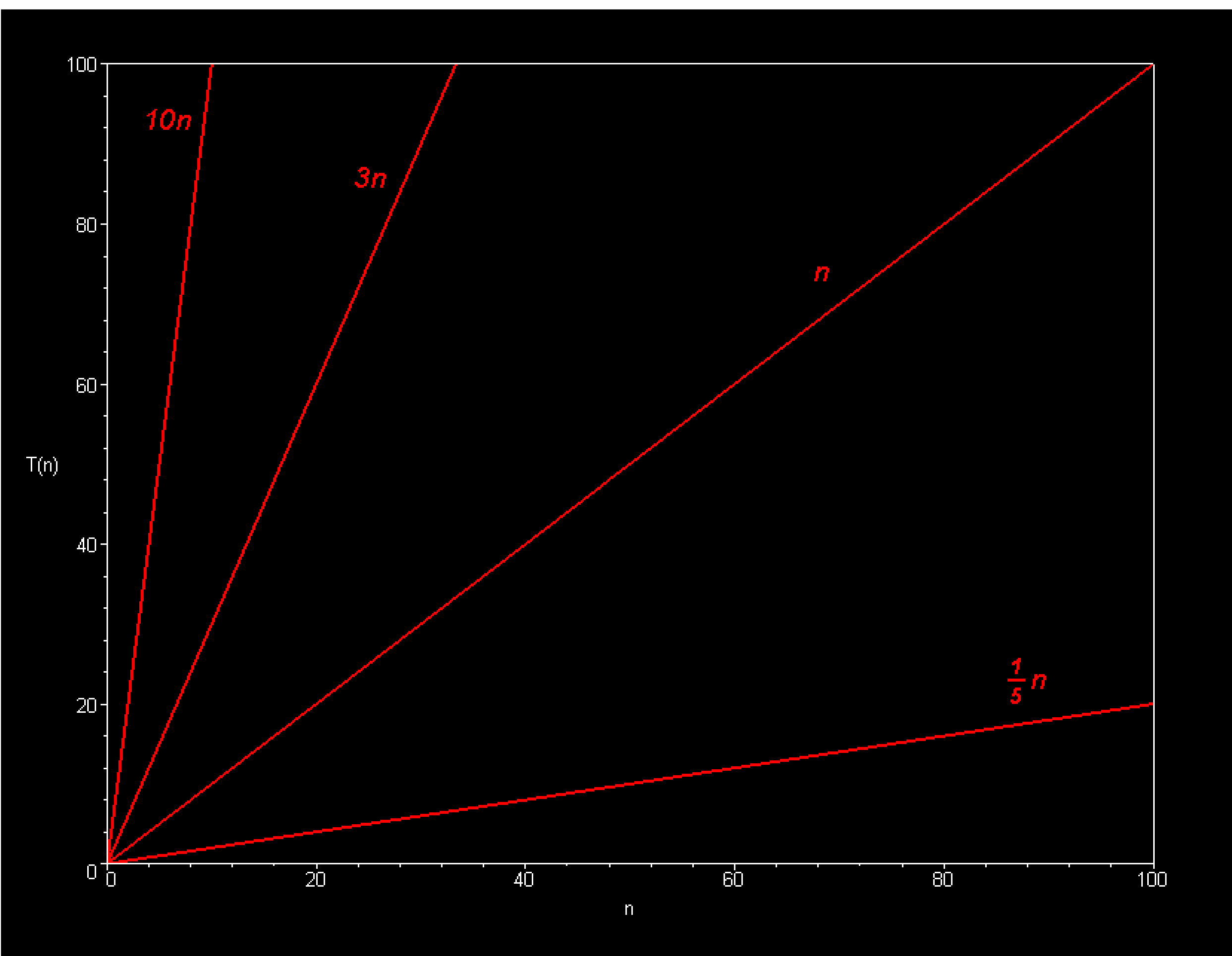


Fig. 5b - Linear Complexity

Quadratic

An Algorithm runs in a quadratic time when execution time is directly proportional to the square of input size. Example - Bubble sort, selection sort

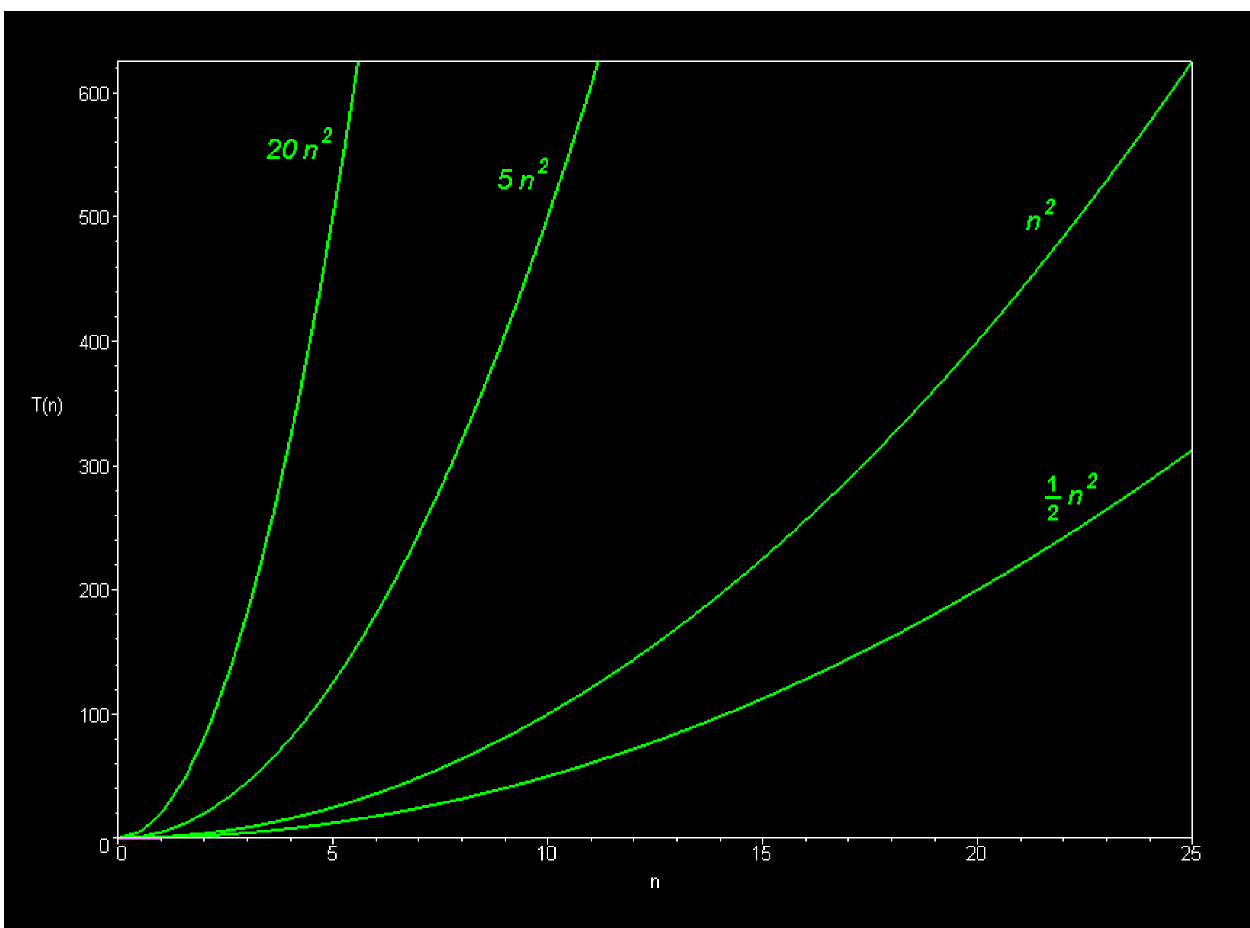


Fig. 5c - Quadratic Complexity

Cubic

An Algorithm runs in a cubic time when execution time is directly proportional to the cube of input size.

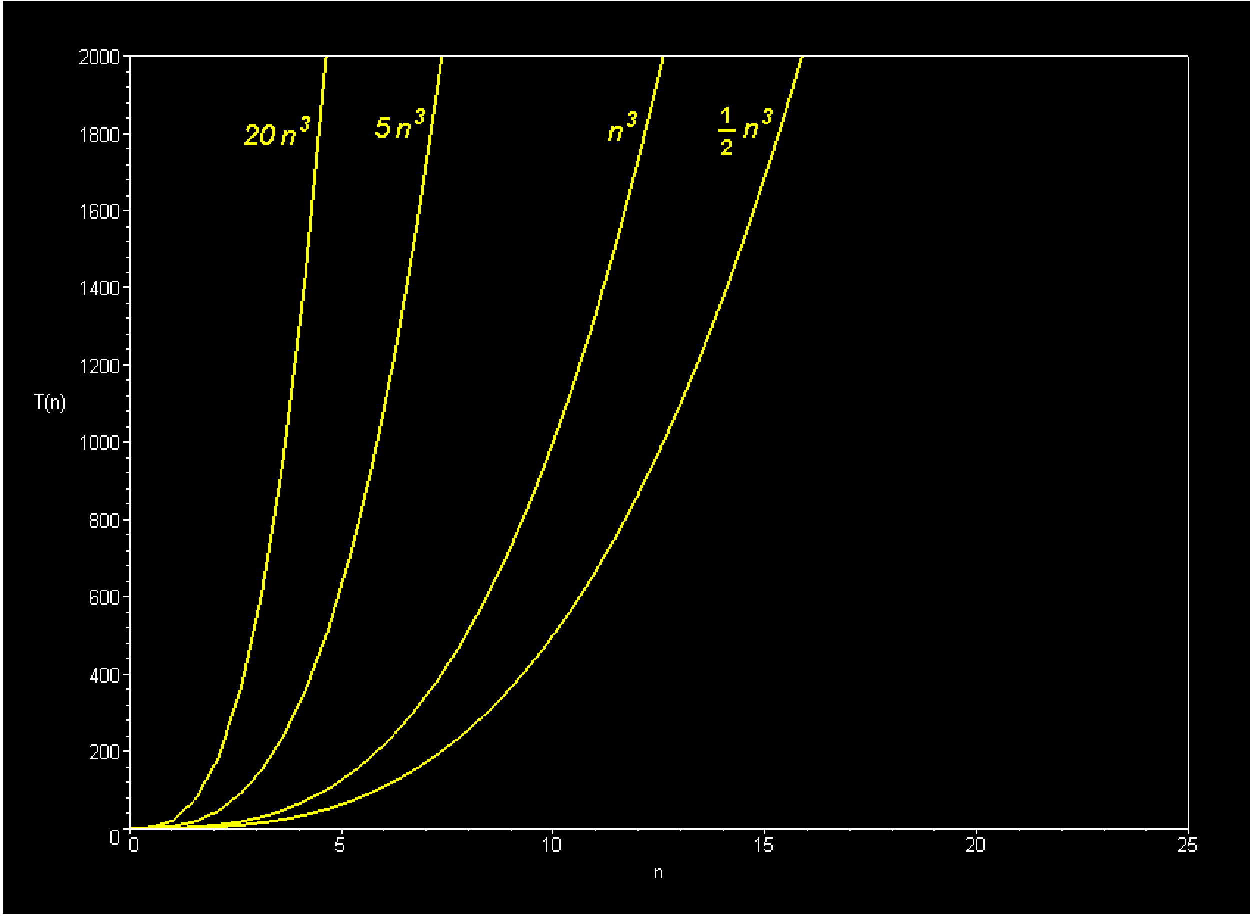


Fig. 5d - Cubic Complexity

Exponential

An algorithm runs in exponential time, when the execution time increases exponentially with the increase in the input size.

Example - Wheat and chessboard problem

Logarithmic

An algorithm runs in logarithmic time, when the execution time is proportional to the logarithm of the input size.

Example - Binary search

Log - Linear

An algorithm runs in log linear time, when the execution time is proportional to n times the logarithm of the input size.

Calculating Complexities

Aside from estimating runtime and space complexity of a program through an analysis of its algorithms, you can understand a program's requirements through experiments. For example. You can measure how long the program takes to run as you increase the size of inputs, data objects, or data files. Use the system.time() function to time calls to functions. Once timing measurements are obtained, time can be plotted against input size and a regression curve can be fitted against the data.

Consider this simple function that sums a list of numeric objects:

```
1  # add the numbers in the vector
2  addNums<-function (v) {
3  l <-length(v)
4  s <-0
5  for (i in 1:l) {
6  s <-s + v[i]
7  }
8  return (s)
9  }
10 s <-addNums(c(3,5,7,1,8,2,3))
11 s
```

To evaluate the running time of an algorithm, we will simply ask how many "steps" it takes. In this case, we can count the number of times it performs the addition. For a vector with n elements, it takes n steps, therefore this function has a time complexity of O(n).

An actual measurement of the time in milliseconds can be calculated using system.time().

```
1  > system.time(s <-addNums(seq(from=1,to=1000000)))
2
3  user system elapsed
4
5  0.64 0.00 0.64
6
7  > system.time(s <-addNums(seq(from=1,to=2000000)))
8
9  user system elapsed
10
11 1.23 0.00 1.23
```

Notice how a vector of twice the size takes about twice as long to be summed.

The code below is written in markdown language. To run this code, open a new R markdown file in Rstudio and copy this code and then paste it in the buffer. Make changes to the code as instructed in the comments, in order to run and check the complexity of your own function.

Note - To run the code below you need to remove one backtick whenever it is mentioned in the comments.

```
1  ---
2  title: "Report"
3  author: "Martin Schedlbauer & Yatish Jain"
4  date: "June 10, 2015"
5  output: html_document
6
7  ---
8
9  ````{r} #Remove one backtick to run the program.
10 #DATA INTAKE PART#
11 ##############################################################################
12 ### In this part of the code replace the functions ###########################
13 ### for which you want to measure complexity. ################################
14 ##############################################################################
15 ### Comment any print line from the function #################################
16 ### to avoid unnecesary data in report #######################################
17 ##############################################################################
18 ### Try to keep the variable to read the file as dataFrame itself. ###########
19 ### Just replace the functions in this file with your functions to ###########
20 ### make this code run. ######################################################
21 ##############################################################################
22
23 addNums<-function (v) {
24 l <-length(v)
25 s <-0
26 for (i in 1:l) {
27 s <-s + v[i]
28 }
29 return (s)
30 }
31
32 ```` #Remove one backtick to run the program
33
34
35 ````{r} #Remove one backtick to run the program
36 #this part of code makes the model graphs
37 n<-c(1:30)
38 t<-100*n
39  t1<-2^n
```

```
40  t2<- ((1/2)*(n^3))
41  t3<-5*(n^2)
42  t4<- 400*sqrt(n)
43  t5<- 400*log(n)
44
45 plot(n,t,type="l")
46
47 text(28,2900,"100n Linear")
48 points(t1,col="blue",type="l")
49 text(10,2980,"2^n Exponential",col="blue")
50 points(t2,col="red",type="l")
51 text(15,2750,"1/3(n^3) Cubic",col="red")
52 points(t3,col="green",type="l")
53 text(23,2600,"5(n^2) Quadratic",col="Dark green")
54 points(t4,col="grey",type="l")
55 text(28,2300,"400sqrt(n) Square root",col="Dark grey")
56 points(t5,col="brown",type="l")
57 text(25,1500,"400log(n) Logarithmic",col="brown")
58
59
60 ```` #Remove one backtick to run the program
61
62 Model complexity Graph which can be used to depict the complexity
63 of your function
64
65
66 ````{r, echo=FALSE} #Remove one backtick to run the program
67
68 ##REPORT GENERATION##
69
70  # Here read the datafile to pass to the function above.
71 data<- seq(from=1,to=1000000)
72
73 double<-rbind(data,data)
74 quad<-rbind(double,double)
75 eight<-rbind(quad,quad)
76 sixteen<-rbind(eight,eight)
77 datafile<-list(data,double,quad,eight,sixteen)
78
79  # change nrow value based on the number of functions you are testing.
80 time<-matrix(nrow=1,ncol=length(datafile))
81
```

```
82  #uncomment the time rows based on the number of functions testing.
83  for(i in 1:length(datafile)){
84    time[1,i]<-system.time(addNums(datafile[[i]]))[3]
85    #time[2,i]<-round(system.time(exampleFunction(data[[i]]))[3],digit=2)
86    #Add as many functions as you want
87    #time[3,i]<-round(system.time(dummyFunction(data[[i]]))[3],digit=2)
88    #time[4,i]<-round(system.time(anything(data[[i]]))[3],digit=2)
89
90  }
91
92  size<-c(1,nrow(double),nrow(quad),nrow(eight),nrow(sixteen))
93  # Instead of 1 use nrow(data) when you are using
94  #an input file with many rows.
95  #Here data is small so I am using 1(hardcoded)
96
97  correlation<-rep(NA,5)
98  #for(i in 1:4) {  Run this loop for the number of
99  #functions you are checking,
100 #example if you are checking 4 functions then run this loop 4 times
101   plot(time[1,]~size, xlab="Data size", ylab="Time (mm)")
102   correlation[1]<-cor(time[1,],size)
103   fit<-lm(time[1,]~size)
104   abline(fit)
105 #}
106
107
108 ```` #Remove one backtick to run the program
109
110 #Graphs of input size vs time for all the different functions used.
111 #Compare the graphs with the model graph to depict the complexity of your functi\
112 on.
```

Explanation

To make the above code work, you must pass the same data values as arguments to all the functions, which you are doubling or quadrupling in the report generation part of the code.

A simple method for doubling or quadrupling the data is to do rbind on data. Another way to do the same thing is by using a loop to send the same data to the function twice or four times. Once we increase the size, we want to make a matrix to store the system time value. For each function, we test as rows and for each complexity we test as columns. For example if you are testing the system time of 4 functions by single data, double data and quadruple data, then you will have to make a matrix of 4 by 3 (4 functions and 3 data sizes).

In the first for loop, we are storing the values of each function to the same row, by keeping the row number the same and increasing the column number by a loop counter. So at the end of this for loop, we have a matrix with each row representing one function and the corresponding column values will have the system time for the respective data sizes. With this matrix, we can plot the graphs for each function using the row of that function in the for loop and fit the regression line over the same.

After editing the above code with your own function, just click on knit HTML to get a report like this.

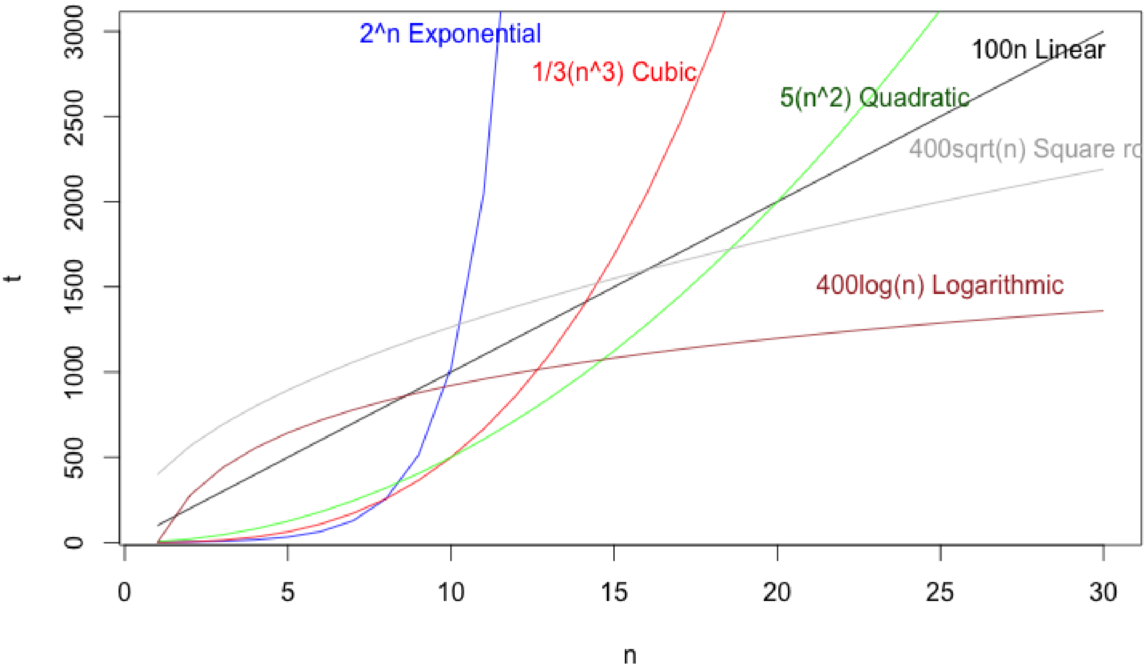
Report

Martin Schedlbauer & Yatish Jain

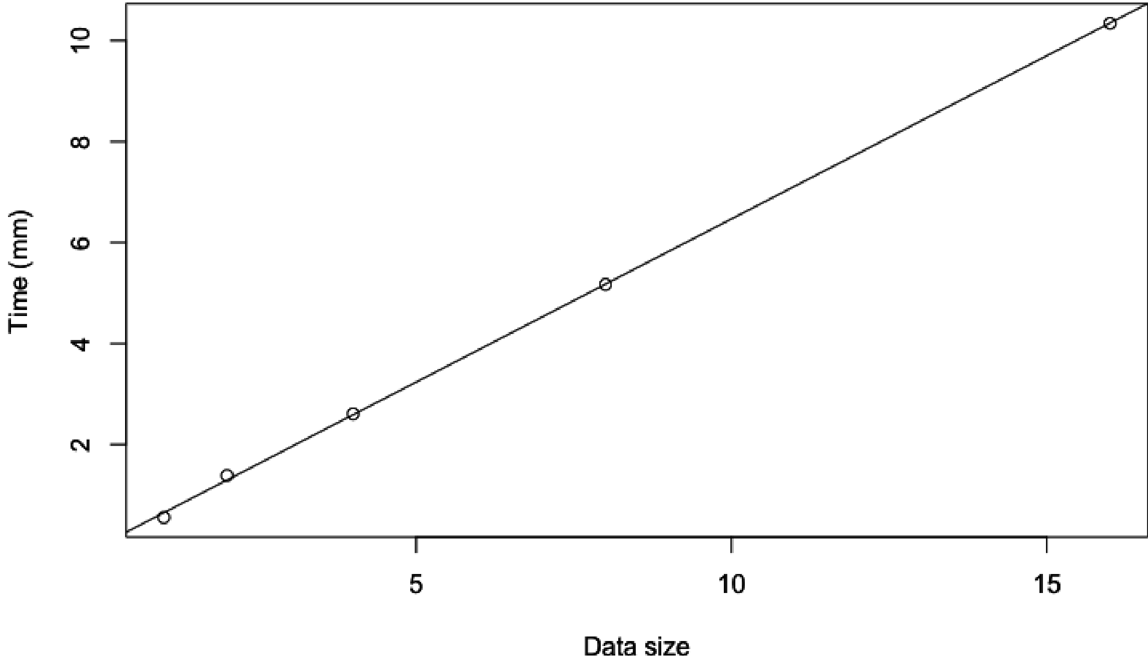
June 10, 2015

```
1  #DATA INTAKE PART#
2  ################################################################################\
3  ################
4  ### In this part of the code replace the functions for which you want to measure\
5   complexity.####
6  ################################################################################\
7  ################
8  ### Comment any print line from the function to avoid unnecesary data in report#\
9  ####
10 ################################################################################\
11 ################
12 ### Try to keep the variable to read the file as dataFrame itself. Just replace \
13 the functions in this file with your functions to make this code run###
14 ################################################################################\
15 ################
16 addNums<-function (v) {
17 l <-length(v)
18 s <-0
19 for (i in 1:l) {
20 s <-s + v[i]
21 }
22 return (s)
23 }
```

```
1  n<-c(1:30)
2  t<-100*n
3   t1<-2^n
4   t2<- ((1/2)*(n^3))
5   t3<-5*(n^2)
6   t4<- 400*sqrt(n)
7   t5<- 400*log(n)
8
9  plot(n,t,type="l")
10 text(28,2900,"100n Linear")
11 points(t1,col="blue",type="l")
12 text(10,2980,"2^n Exponential",col="blue")
13 points(t2,col="red",type="l")
14 text(15,2750,"1/3(n^3) Cubic",col="red")
15 points(t3,col="green",type="l")
16 text(23,2600,"5(n^2) Quadratic",col="Dark green")
17 points(t4,col="grey",type="l")
18 text(28,2300,"400sqrt(n) Square root",col="Dark grey")
19 points(t5,col="brown",type="l")
20 text(25,1500,"400log(n) Logarithmic",col="brown")
```



Model complexity Graph which can be used to depict the complexity of your function



Graphs of input size vs time for all the different functions used. Compare the graphs with the model graph to depict the complexity of your function.

Chapter 6

Importing data in R

R has a very wide variety of packages available that allows import of almost any format. The most common formats are:

File Format	Extension	Source	Format
Comma Separated	.csv	Excel; many programs export data in that format	Text
Tab Delimited	.txt	many programs export data in that format	Text
Excel	.xls	Excel files prior to version 2010	Encoded
Excel	.xlsx	Excel version 2010 and later	XML
R Data Object	.RData	R	Binary

Fig. 6a - Different Formats

Other than text and R binary files, R can also import data from other statistical packages using the "foreign" library. Among many others, the "foreign" library supports importing from:

- Stata - read.dta() function to read and write.dta() function to write.
- SPSS - read.spss() function to read and write.foreign() function to write
- SAS - read.xport() function to read and write.foreign() function to write

The first step to import data is to set the working directory to the folder where your file is located. To know the current working directory getwd() function is used and to change the working directory setwd() function is used.

```
1 > ## absolute path to a folder
2 > setwd("C:/Users/Martin/Downloads")
3 > ## relative path to a folder
4 > setwd("../")
```

The setwd() function follows basic Linux commands of taking both absolute and relative paths as arguments. In a relative path few basic symbols are:

1. ~ Home directory
2. . current working directory
3. .. one directory up from current directory

R for Windows understands the forward slash but not a single backslash, i.e., this specification will not work:

```
1 > ## will not work
2 > setwd("C:\Users\Martin\Downloads")
```

But this will work:

```
1 > ## will  work
2 > setwd("C:\\Users\\Martin\\Downloads")
```

Defensive programming

Its always a good habit to incorporate defensive programming techniques when programming. When importing a file , it is good practice to check whether the file exists in the current working directory or not. The file.exists() function checks if the file or directory exists and returns TRUE if it does, FALSE otherwise.

Another similar function is the exists() function, it check if the object exists in the current environment. Usage of the exists() function is very common, expecially when importing big data, since you do not want to import data that already exists. When a file is loaded, its contents are stored in the computer's RAM (main memory) which is often limited. The 64-bit version of R can hold more data than the 32-bit version of R. Aside from in-memory storage, files can also take significant time to load.

The dir.create() function: creates the provided passed directory, if it does not already exist.

```
> ## create a new directory (folder) if it
> ## does not exist
> if (!file.exists("data")) {
+ dir.create("data")
+ }

> if(!exists("dataset")){
+ dataset<- read.table(bigDataFile, header=TRUE, sep=" ")
+ }
```

The above code becomes very important when dealing with big data as we don't want to load the dataset everytime we run our code as that will cost us both time and memory. We should always follow this practice of defensive programming when working with big data.

File Import

Fixed width files

Fixed width text files are a special class of text files where the format is specified with column width, alignments and padded characters. Data is arranged in rows and columns with one entry per row. Column width is constant throughout the file and is defined by character, which in turn determines the amount of data it can contain.

Below is an example of a file with a fixed width. The complete file can be accessed using this link Weekly SST data[8]

```
 Weekly SST data starts week centered on 3Jan1990

                Nino1+2      Nino3        Nino34        Nino4
 Week          SST SSTA     SST SSTA     SST SSTA     SST SSTA
 03JAN1990     23.4-0.4     25.1-0.3     26.6 0.0     28.6 0.3
 10JAN1990     23.4-0.8     25.2-0.3     26.6 0.1     28.6 0.3
 17JAN1990     24.2-0.3     25.3-0.3     26.5-0.1     28.6 0.3
 24JAN1990     24.4-0.5     25.5-0.4     26.5-0.1     28.4 0.2
```

Fixed width files can be read using read.fwf() function. This function reads the files directly into a data frame.

[8] http://www.cpc.ncep.noaa.gov/data/indices/wksst8110.for

```
1  > x <- read.fwf(file=url("http://www.cpc.ncep.noaa.gov/data/indices/wksst8110.fo\
2  r"),skip=4,widths=c(12, 7,4, 9,4, 9,4, 9,4))
3  > head(x)
4              V1   V2   V3   V4   V5   V6   V7   V8  V9
5  1  03JAN1990   23.4 -0.4 25.1 -0.3 26.6  0.0 28.6 0.3
6  2  10JAN1990   23.4 -0.8 25.2 -0.3 26.6  0.1 28.6 0.3
7  3  17JAN1990   24.2 -0.3 25.3 -0.3 26.5 -0.1 28.6 0.3
8  4  24JAN1990   24.4 -0.5 25.5 -0.4 26.5 -0.1 28.4 0.2
9  5  31JAN1990   25.1 -0.2 25.8 -0.2 26.7  0.1 28.4 0.2
10 6  07FEB1990   25.8  0.2 26.1 -0.1 26.8  0.1 28.4 0.3
11
12 > str(x)
13 'data.frame':        1332 obs. of  9 variables:
14  $ V1: Factor w/ 1332 levels " 01APR1992  ",..: 102 409 715 1021 1314 275 582 88\
15 8 1194 290 ...
16  $ V2: num  23.4 23.4 24.2 24.4 25.1 25.8 25.9 26.1 26.1 26.7 ...
17  $ V3: num  -0.4 -0.8 -0.3 -0.5 -0.2 0.2 -0.1 -0.1 -0.2 0.3 ...
18  $ V4: num  25.1 25.2 25.3 25.5 25.8 26.1 26.4 26.7 26.7 26.7 ...
19  $ V5: num  -0.3 -0.3 -0.3 -0.4 -0.2 -0.1 0 0.2 -0.1 -0.2 ...
20  $ V6: num  26.6 26.6 26.5 26.5 26.7 26.8 26.9 27.1 27.2 27.3 ...
21  $ V7: num  0 0.1 -0.1 -0.1 0.1 0.1 0.2 0.3 0.3 0.2 ...
22  $ V8: num  28.6 28.6 28.6 28.4 28.4 28.4 28.5 28.9 29 28.9 ...
23  $ V9: num  0.3 0.3 0.3 0.2 0.2 0.3 0.4 0.8 0.8 0.7 ...
```

Writing a fixed width file can be done by write.fwf() function which is provided in the gdata package.

```
1  >install.packages("gdata")
2  >library("gdata")
3  > num<-round(rnorm(10),2)
4
5  > testData <- data.frame(num1=c(1:10, NA),
6  + num2=c(NA, seq(from=1, to=5.5, by=0.5)),
7  + num3=c(NA, num),
8  + int1=c(as.integer(1:4), NA, as.integer(4:9)),
9  + fac1=factor(c(NA, letters[1:9], "hjh")),
10 + fac2=factor(c(letters[6:15], NA)),
11 + cha1=c(letters[17:26], NA),
12 + cha2=c(NA, "longer", letters[25:17]),
13 + stringsAsFactors=FALSE)
14
15 > write.fwf(testData, na="NA" )
16
```

```
17  num1 num2 num3 int1 fac1 fac2 cha1 cha2
18   1   NA    NA 1 NA   f  q   NA
19   2 1.0 -1.75 2 a    g  r   longer
20   3 1.5 -1.41 3 b    h  s   y
21   4 2.0  2.12 4 c    i  t   x
22   5 2.5  0.79 NA d    j  u   w
23   6 3.0  0.33 4 e    k  v   v
24   7 3.5 -0.72 5 f    l  w   u
25   8 4.0  0.49 6 g    m  x   t
26   9 4.5  0.30 7 h    n  y   s
27  10 5.0  0.12 8 i    o  z   r
28  NA 5.5 -1.12 9 hjh NA NA q
```

Unstructured text files

Many times, a data analyst is given a file that does not follow a predefined format. To deal with this type of file, you have to read it in such a way that becomes conducive to analysis. Functions like scan() and readLines() can scan/ read your data word by word. The readLines() function provides the encoding argument; the encoding argument specifies the encoding of the file.

```
1  data<- readLines("myFile.txt", encoding="UTF-8")
2
3  scan("myFile.txt", character(0)) # separate each word
4  scan("myFile.txt", character(0), sep = "\n") # separate each line
```

CSV

CSV(Comma separated values) files contain data records organized in rows with an optional header row containing the column labels. There is no standard CSV file format, although RFC 4180 is an attempt to standardize some aspects of a .csv file format. One optional aspect of a .csv file, is that data values can be optionally double quoted. This allows the comma character to be present in a data value. Data values can also be missing.

```
1  #Example CSV file
2
3  "first_name","last_name","company_name","address","city","county","state","zip",\
4  "phone1","phone2","email","web"
5
6  "James","Butt","Benton, John B Jr","6649 N Blue Gum St","NewOrleans","Orleans","\
7  LA",70116,"504-621-8927","504-845-1427","jbutt@gmail.com","http://www.bentonjohn\
8  bjr.com"
9
```

```
10 "Josephine","Darakjy","Chanay, Jeffrey A Esq","4 B Blue Ridge Blvd","Brighton","\
11 Livingston","MI",48116,"810-292-9388","810-374-9840","josephine_darakjy@darakjy.\
12 org","http://www.chanayjeffreyaesq.com"
```

CSV files can be loaded in R using the read.csv() function. The function takes many arguments, the list below provides a description for the more popular arguments :

- Header - If the file contains a header row with column labels, specify header = TRUE else FALSE.
- Skip - If the data of interest starts anywhere but the first row, specify skip=x, where x is number of lines to be skipped.
- Sep - While CSV files are expected to use comma (,) as a separator, some might not. The sep="x" allows you to specify any character as the field separator.
- quote - In both CSV and tab delimited flat files, it is not uncommon that data values are placed inside quotes: " or '. Specifying quote="" causes the quotes to not be read. If data values are placed within non-standard quotes, then the data values must be processed as character strings and the quotes must be removed through programming.
- stringsAsFactors - Strings are often converted to factors which may not be the desired result. Specify stringsAsFactors=FALSE to avoid having R automatically convert string values to a factor.

```
1  > data<- c("Name",1,2,3,"text")
2  > data1<-c("John",34,12,65,"Hello world")
3  > data2<- c("Michelle",21,87,98,"Good day")
4
5  > df<-data.frame(data,data1,data2)
6
7  > write.csv(df, file="myData.csv")
8  > import<-read.csv("myData.csv",header=T, sep=",",stringsAsFactors=F)
9  > import
10   X data       data1    data2
11 1 1 Name        John Michelle
12 2 2    1          34       21
13 3 3    2          12       87
14 4 4    3          65       98
15 5 5 text Hello world Good day
```

A tab delimited file

A tab delimited file is the same as a CSV file except that the separator is a tab (the \t character in R). Reading a tab-delimited file can be done with:

- read.csv() using sep="\t"
- read.table()

```
1  > people <-read.table("us-500.txt",header=TRUE)
2  > str(people)
```

Excel

This file format is the native format for Excel. Excel also exports data as CSV, but sometimes data is only available in the native file format. Excel files have a .xls and .xlsx extension. Importing such a file requires the xlsx library.

Excel files can contain one or more worksheets, so you must specify which sheet is to be loaded using sheetIndex=x where x is the number of the worksheet.

```
1  > library(xlsx)
2  > people <-read.xlsx("us-500.xlsx",sheetIndex=1)
3  > str(people)
```

> *Note* - This may require the use of the 32-bit version of R.

Data can be written to an Excel file using the function write.xlsx(). However, for compatibility, store data in CSV or tab-delimited text files rather than in Excel format.

```
1  > library(xlsx)
2  > data<- c("Name",1,2,3,"text")
3  > data1<-c("John",34,12,65,"Hello world")
4  > data2<- c("Michelle",21,87,98,"Good day")
5  > df<-data.frame(data,data1,data2)
6
7  > write.xlsx(df, file="myData.xlsx")
8  > import<-read.xlsx("myData.xlsx",sheetIndex=1)
9  > import
10   NA. data       data1    data2
11 1   1 Name        John Michelle
12 2   2    1          34       21
13 3   3    2          12       87
14 4   4    3          65       98
15 5   5 text Hello world Good day
```

The xlsx package is slow and time consuming. When dealing with a large file, alternative packages like openxlsx and XLConnect are fast and offer many alternative functions for writing data.

```
1  > library("XLConnect")
2  > wb = loadWorkbook("xlconnect1.xlsx",create=T)  # create if doesn't exist
3
4  > createSheet(wb,"cars stats")
5  > writeWorksheet(wb,cars,"cars stats")
6
7  > saveWorkbook(wb)
8
9  > data<- readWorksheet(wb,"cars stats",header=T)
10 > str(data)
11 'data.frame':        50 obs. of  2 variables:
12  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
13  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
14
15
16 library(openxlsx) #load package
17 #read file into data frame accounting for 3rd row as starting row, essentially o\
18 mmiting unnecessary huge title and description
19 data<-read.xlsx("DATA.xlsx", sheet=1, startRow=3)
```

Binary R data

Saving R objects in .RDatafiles is fast and convenient but not compatible with programs other than R. Use the load() function to load previously save objects. To determine which objects were loaded, use the objects() or ls() commands.

```
1  > ls()
2   [1] "addNums"          "char.vec"         "correlation"      "data"
3   [5] "data2"            "datafile"         "date"             "date1"
4   [9] "df"               "double"           "dt1"              "dt2"
5  [13] "fac"              "fit"              "i"                "import"
6  [17] "n"                "num"              "num.vec"          "pdf"
7  [21] "s"                "sixteen"          "size"             "t"
8  [25] "t3"               "t4"               "t5"               "text"
9  [29] "tm1"              "tm1.lub"          "tm2"              "tm2.lub"
10 [33] "x"                "xit"
11
12 > save.image('Data.RData')
13 > rm(list=ls())
14 > ls()
15
16 character(0)
17
```

```
18 > load('Data.RData')
19 > ls()
20
21  [1] "addNums"      "char.vec"     "correlation"  "data"
22  [5] "data2"        "datafile"     "date"         "date1"
23  [9] "df"           "double"       "dt1"          "dt2"
24 [13] "fac"          "fit"          "i"            "import"
25 [17] "n"            "num"          "num.vec"      "pdf"
26 [21] "s"            "sixteen"      "size"         "t"
27 [25] "t3"           "t4"           "t5"           "text"
28 [29] "tm1"          "tm1.lub"      "tm2"          "tm2.lub"
29 [33] "x"            "xit"
```

Rdata files are of most importance when you are dealing with big data since loading of big data is time consuming. To load multiple large files, sometimes there is a need to clear the cache variables as the content is stored in RAM. In such cases, it is good practice to save the loaded data first to an Rdata image. The advantage of using an Rdata object is that objects in that session can be restored without actually loading the data again thus saving a lot of time.

XML

XML stands for Extensible Markup Language. It was designed to describe data in a human readable format that is simple to parse. The purpose of XML is to provide a software and hardware independent encoding format for carrying information. The data is described within XML in the form of a tree.

An XML document consists of matching nested tags describing data. The tags are free-form and require the sender and receiver of the document to agree upon their meaning and representation. There are numerous industry-standard XML schemas.

```
1 # Example XML code
2 <?xml version="1.0" encoding="UTF-8"?>
3 <Student>
4 <FName>John</FName>
5 <LName>Doe</LName>
6 <Mark>60.0</Mark>
7 <Grade>A </Grade>
8 </Student>
```

XML documents must begin with a declaration that specifies information needed by the parser. The general declaration looks like this:

```
<?xmlversion="1.0" encoding="UTF-8"?>
```

The tag is a markup construct that begins with < and ends with >. Tags come in three flavors:

- start tag: <Student>
- end tag: </Student>
- empty-element tag: <isActive/>

Every start tag must have a matching end tag or the document will not parse. The characters between the start and end tags, if any, are the element's content.

Tags nested within other tags are referred to as child elements.

```
<!--
attributes can be expressed in any object's start tag  and follow the format
attributename=attrbutevalue
-->
<course crn="3387">
     <title>Programming in R</title>
     <program>Data Science</program>
     <instructor>
     <name>Martin Schedlbauer, Ph.D.</name>
     <email>m.schedlbauer@neu.edu</email>
     </instructor>
</course>
```

In the above example, the title, program and instructor tags are children elements of the parent tag course. Similarly, Instructor is a parent tag with two child elements name and email.

XML Package

The "XML" package contains the necessary functions to read and parse XML. The XML package provides the functions necessary to load an XML file and parse its document tree:

- xmlParse()
- xmlToDataFrame()

xmlParse()

The xmlParse() function is used to parse the XML data and creates a document object of class XMLInternalDocument.

```
xmlobj<-xmlParse("pubmed_sample.xml")
```

Once the files have been successfully parsed, R stores the XML document in the internal object xmlobj.

xmlToDataFrame()

The xmlToDataFrame() function is used to parse the XML data directly into a dataframe.

XML can be parsed in R in two ways:

- Parsing via HTTP
- Parsing into a tree

Parsing via HTTP

An XML document can also be loaded via HTTP through its URL using either xmlParse() or xmlToDataFrame().

```
> url<-"http://www.statistics.life.ku.dk/primer/mydata.xml"
> data <-xmlToDataFrame(url)

> head(data)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
5  10.7     81   18.8
6  10.8     83   19.7
```

Parsing into a tree

To navigate the document object, R requires parsing with xmlTreeParse() followed by retrieving the root node object using xmlRoot().

```
> xmlobj<-xmlTreeParse("pubmed_sample.xml")
> r <-xmlRoot(xmlobj)
```

r is of class XMLNode.

> *Note* - pubmed_sample.xml file can be downloaded from this link. pubmed_sample.xml[9]

Once the data is parsed into a tree, different functions are used to access the data from the tree:

- xmlName() - get the name of the root element.

[9] https://drive.google.com/file/d/0B9uiGI8JEJw5Xzg3dmk5Tnc5WVk/view

```
> xmlName(r)
[1] "PubmedArticleSet"
```

- xmlSize() - returns the number of subelements (children) for the passed XMLnode

```
> xmlSize(r)
[1] 19
```

Accessing a child node

Each child node is accessible through subscripting through its parent node.

```
> r[[1]]
<PubmedArticle>
<MedlineCitation Owner="NLM" Status="PubMed-not-MEDLINE">
<PMID Version="1">23874253</PMID>
<DateCreated>
...
> xmlName(r[[1]])
[1] "PubmedArticle"
```

Navigating the tree

Repeated subscripting is used to navigate the tree.

```
> r[[1]][[2]][[1]][[1]]
<PubMedPubDatePubStatus="received">
<Year>2012</Year>
<Month>1</Month>
<Day>15</Day>
</PubMedPubDate>
```

So in the above example: The root of the tree is <PubmedArticleSet>, therefore:

- r[[n]]is the nth child of <PubmedArticleSet>
- r[[n]][[k]]is the kth child of the nth node under <PubmedArticleSet>
- and so on…

Node Attributes

Attributes are name/value pairs attached to a start tag. Retrieve the attributes as a vector using xmlAttrs(), then access individual attributes using subscripting.

```
> r[[1]][[2]][[1]][[1]]
<PubMedPubDatePubStatus="received">
<Year>2012</Year>
<Month>1</Month>
<Day>15</Day>
</PubMedPubDate>

> attrs<-xmlAttrs(r[[1]][[2]][[1]][[1]])
> attrs
PubStatus
"received"
```

Values

Once we have discovered the correct node, then the xmlValue() function is used to read the value between the tags.

```
> r[[1]][[1]][[2]]
<DateCreated>
<Year>2013</Year>
<Month>07</Month>
<Day>22</Day>
</DateCreated>

> r[[1]][[1]][[2]][[1]]
<Year>2013</Year>

> xmlValue(r[[1]][[1]][[2]][[1]])
[1] "2013"
```

To get all of the children nodes as a list, use subsetting.

```
> r[[1]][[1]][[2]][1:3]
$Year
<Year>2013</Year>
$Month
<Month>07</Month>
$Day
<Day>22</Day>
attr(,"class")
[1] "XMLNodeList"
```

```
11 > r[[1]][[1]][[2]][1:3]$Year
12 <Year>2013</Year>
```

The xmlSApply() function applies a function over a list or vector. It essentially implements the Visitor pattern (apply a function to each node visited). This example applies the xmlName() function to each child node.

```
1 > xmlSApply(r[[1]], xmlName)
2   MedlineCitation        PubmedData
3 "MedlineCitation"      "PubmedData"
```

Chapter 7

Web Scraping

As the Internet continues to grow, the amount of data available has increased substantially. However, availability of data does not translate to accessibility. Web scraping(or web harvesting or web data extraction) extracts data from websites when the data is not available in text file format, such as CSV.

Example: YellowPages does not make its data available as a file or a Web API, so extracting or "scraping" the data from their websites is required. To extract a list of the restaurants use a GET request to specify the search and then "parse" or "scrape" the HTML page that is returned. This would be done programmatically.

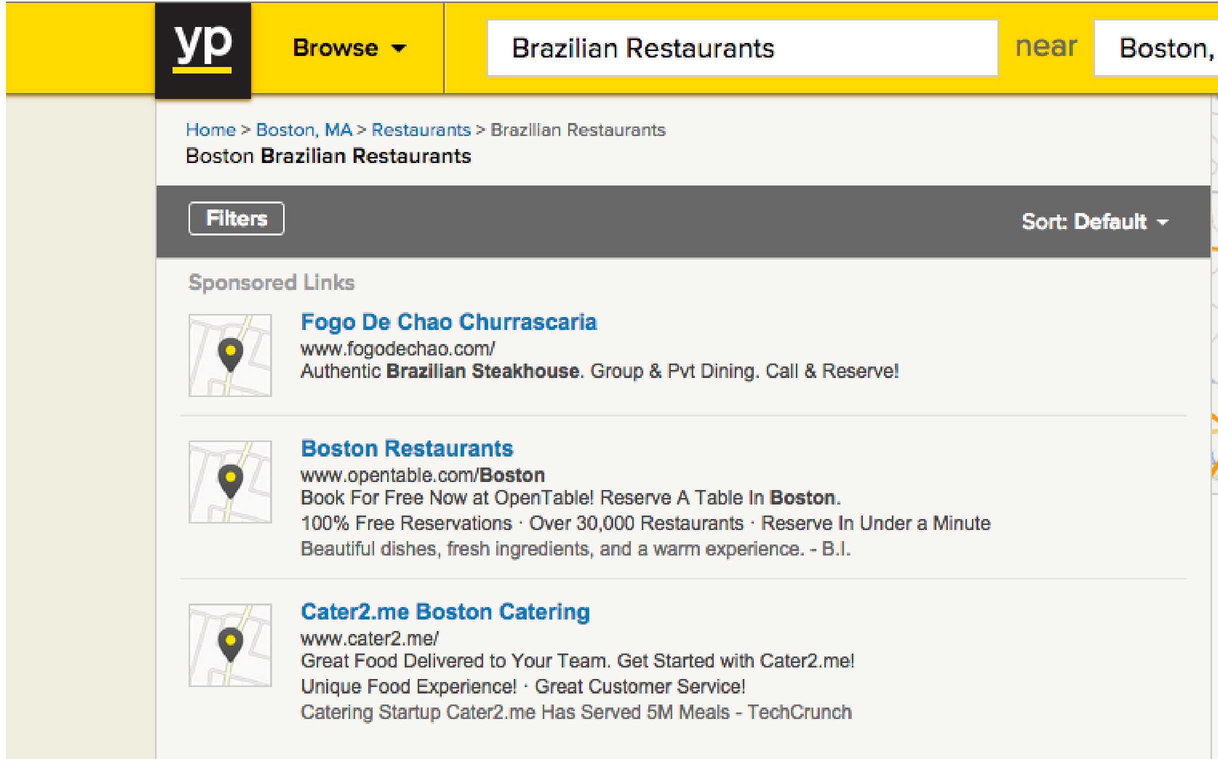


Fig. 7a - Yellow pages example

Although data can be manually copied from websites, it will be too tedious and time consuming for anything but very small amounts of data. "Web Scraping" automates this process, so that instead of manually copying the data from websites, a program does the copying.

Web Scraping essentially transforms the raw unstructured data in HTML into structured data that can be cleaned, stored, shaped, and used in analysis. Once the data is in structured format, it can be easily manipulated for any analysis depending upon the requirement.

Advantages of web scraping services:

- Automation of data
- Data collected is accurate and reliable
- Data can be collected from both static and dynamic pages
- End result of the scraping gives the data in a format conducive for further analysis.

Major challenges of web scraping services:

- Source complexity: Depending upon the complexity of information need to be extracted scraping can be parallelized.
- High volume of scraping can cause regulatory damage to pages.
- Scale of measure: scaling can become an issue depending upon the measure of the source data to be scraped.
- Legal Issues

Legal issues:

Some websites do not allow web scraping as part of their terms of use. Additionally, the scraped data may be private or copyrighted and may be prohibited from being copied or used for commercial purposes.

Legal theories related to automated online data collection:

1. Copyright Infringement
2. Breach of contract
    A. Enforceability of website terms of use.
    B. Terms of use that may prohibit automated data collection
3. Computer fraud and abuse act
4. Hot news misappropriation

Current web scraping approaches range from ad-hoc, manual scraping, to fully automated parsing.

- Manual Copy-and-Paste
- Regular Expression Matching
- **HTTP Retrieval with HTML Parsing**
- DOM Parsing
- **Web Scraping Toolkits**
- Vertical Integration Platforms
- Metadata & Semantic Markup Recognition
- Machine Learning Based Visual Scanning

Web scraping toolkits

There are several web scraping toolkits available:

- Google chrome scraper
- Kimono lab
- Import io
- Simple PHP scraper
- Outwit hub
- ScraperWiki
- Fminer.com

All the scraping toolkits essentially works on the same principle with some advantages or disadvantage over one another.

Google chrome scraper

To install Google chrome scraper, go to chrome web store and search for a scraper, Add the first extension to your Google chrome.

Actual scraping is very easy with this extension, Just select the data on any web page, right click on the selected data and click scrape similar.

Let's see an actual working example for scraping all the links on www.boattrader.com

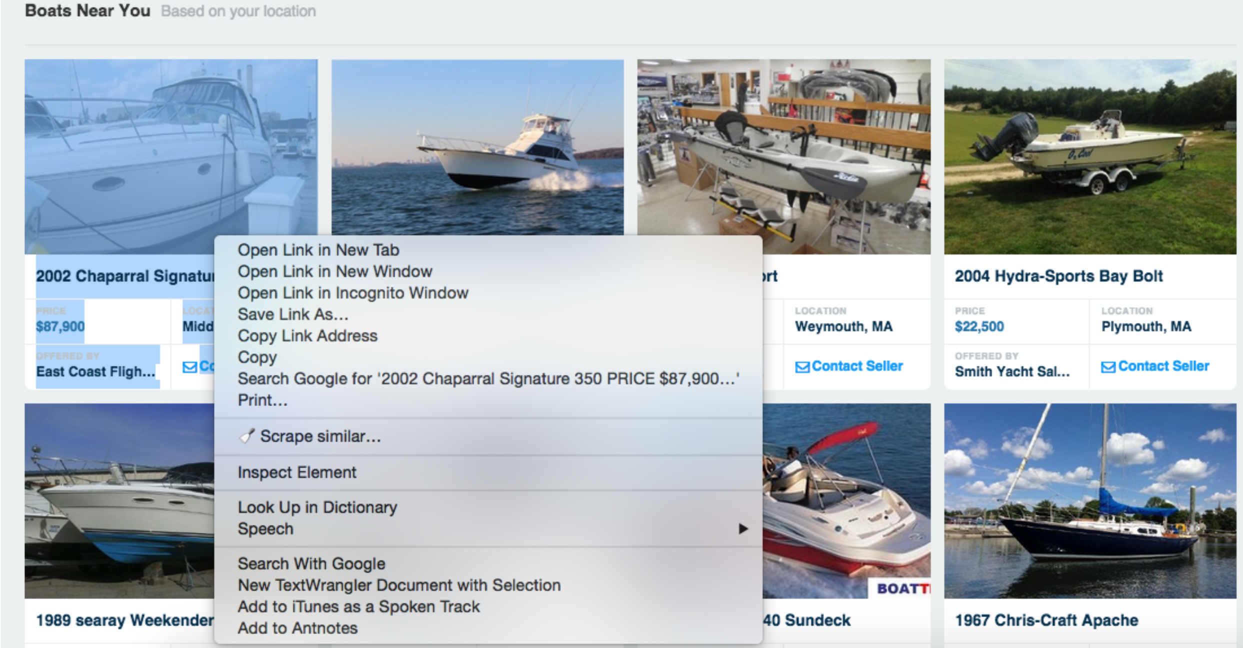


Fig. 7b - Scraping boats link

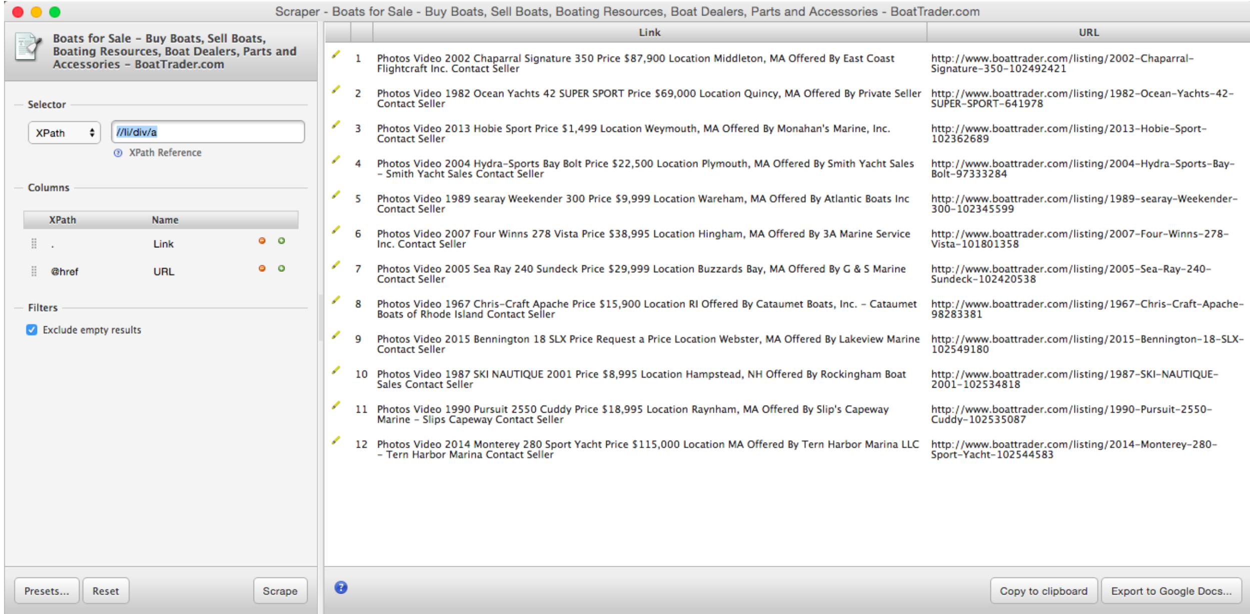


Fig. 7c - Scraped data

As shown in the image above, Google chrome web scraper is a simple tool to use. It makes copying data to a clipboard or to a Google document fairly easy. Any element not accessible easily can be scraped by editing the XPath reference. XPath is a query language for HTML and XML.

Let's see a complex scraping with the use of XPath reference.

We will scrape the data of all the movies of Actor Will Ferrell from IMDB. Will Ferrell[10]

Following the same steps as above:

- Select one of the movies and click scrape similar.

[10]http://www.imdb.com/name/nm0002071/

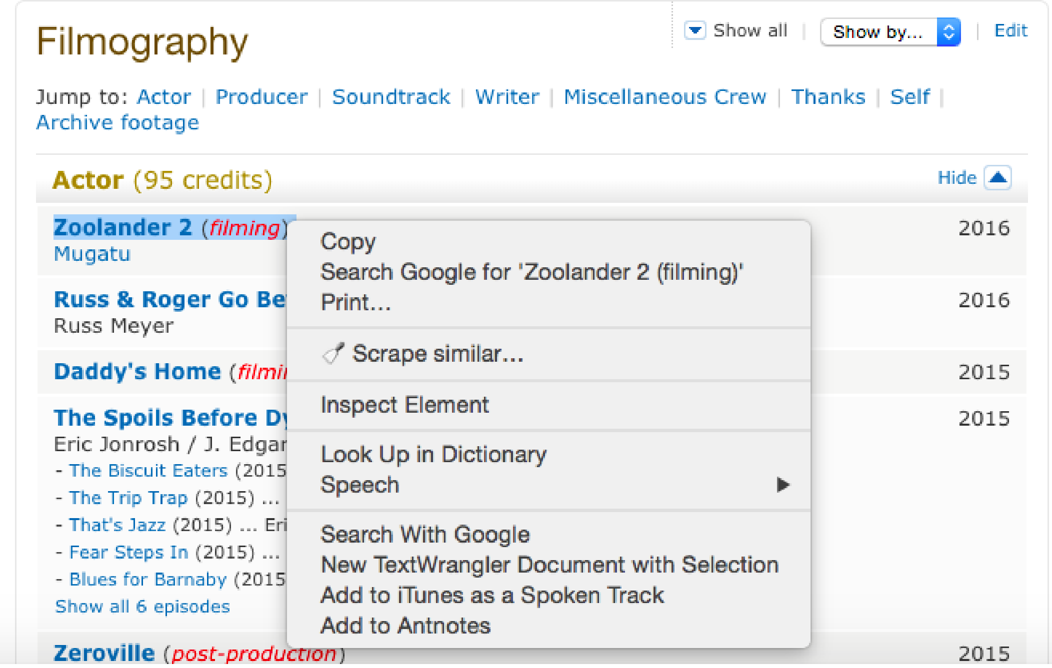


Fig. 7d - Movies of Will Ferrell

- But the data scraped this time is not proper because the list is not structured properly and the scraper cannot distinguish the difference between movie title and year.

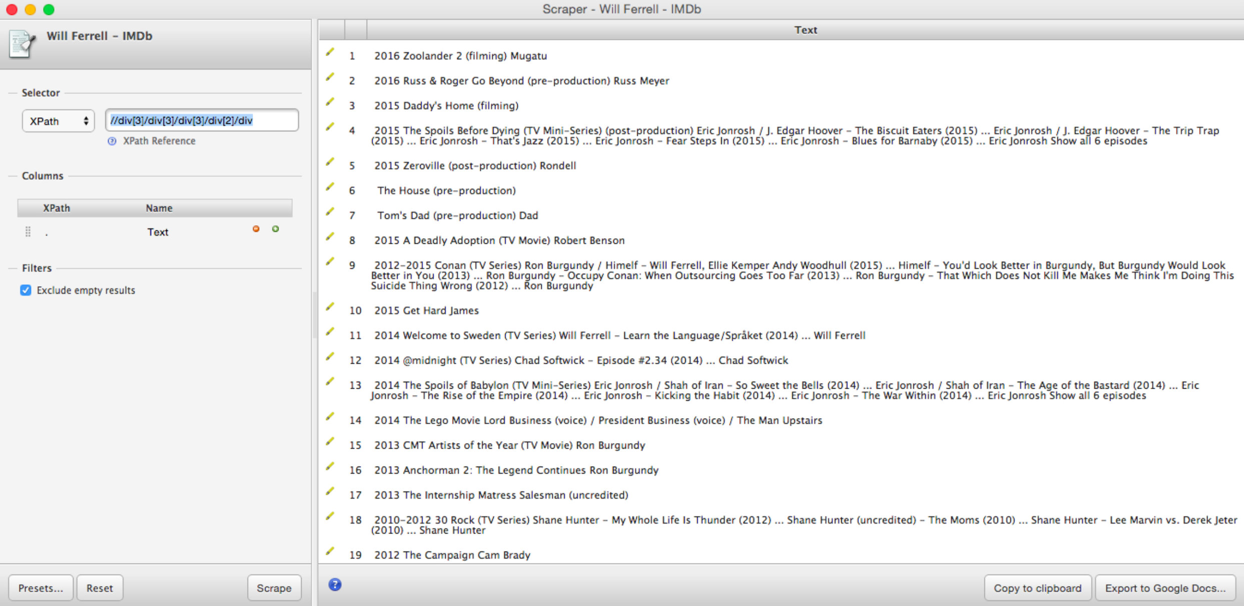


Fig. 7e - Scraped data

- The current XPath reference //div[3]/div[3]/div[3]/div[2]/div, contains the HTML data for this data. RIght click on movie name and click inspect element.

- Look closely for the tags around movie name, here movie name is enclosed within <b> tag.

```
▼<div class="filmo-row odd" id="actor-tt1608290">
   <span class="year_column">
    2016
   </span>
  ▼<b>
     <a href="/title/tt1608290/?ref_=nm_flmg_act_1">Zoolander 2</a>
   </b>
```

Fig. 7f - Movie name HTML

- Let's add a b element in the XPath reference and see the result.

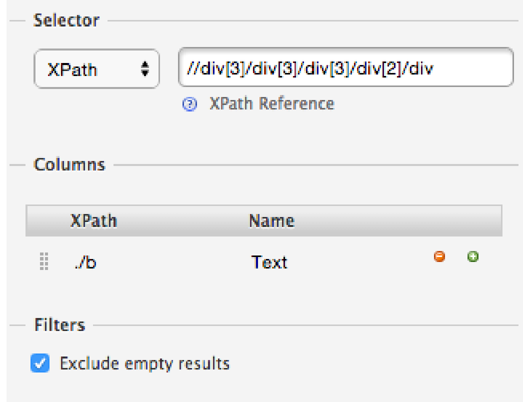


Fig. 7g - Adding b tag to XPath

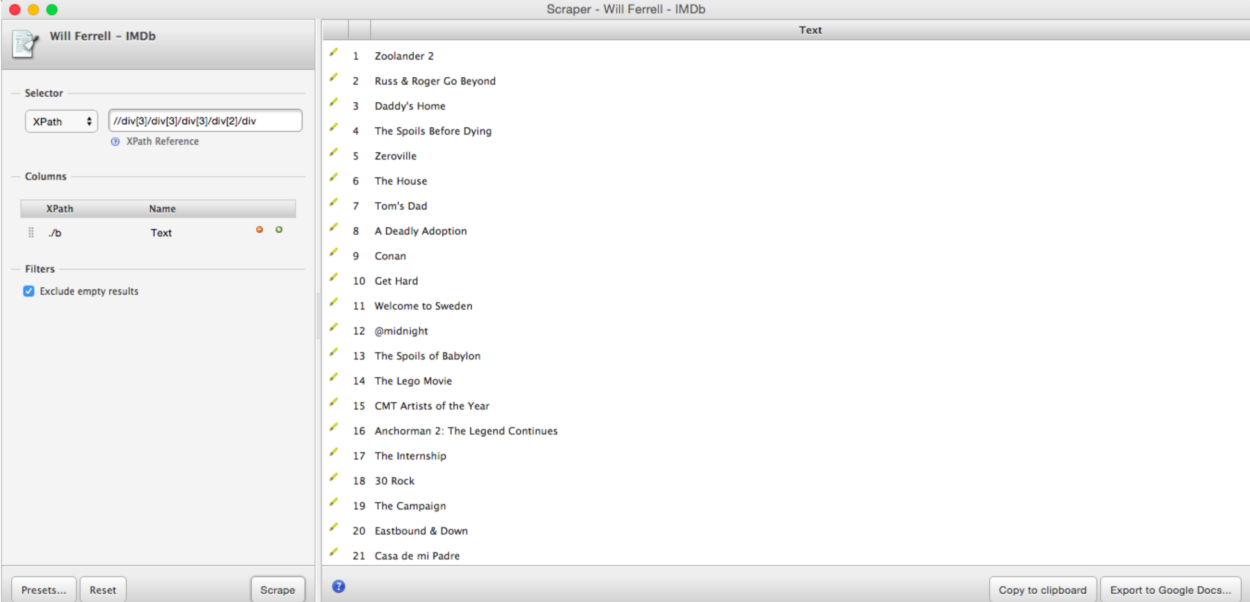


Fig. 7h - Movies scraped

- Further we can scrape year in a separate column. If we look at the HTML in the above image, the year is enclosed within span tag, so we can add span tag to XPath just like b tag to scrape year separately.

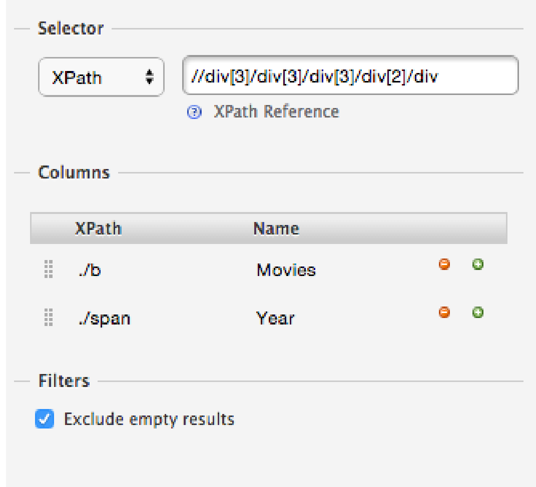


Fig. 7i - Adding span tag to XPath in second column

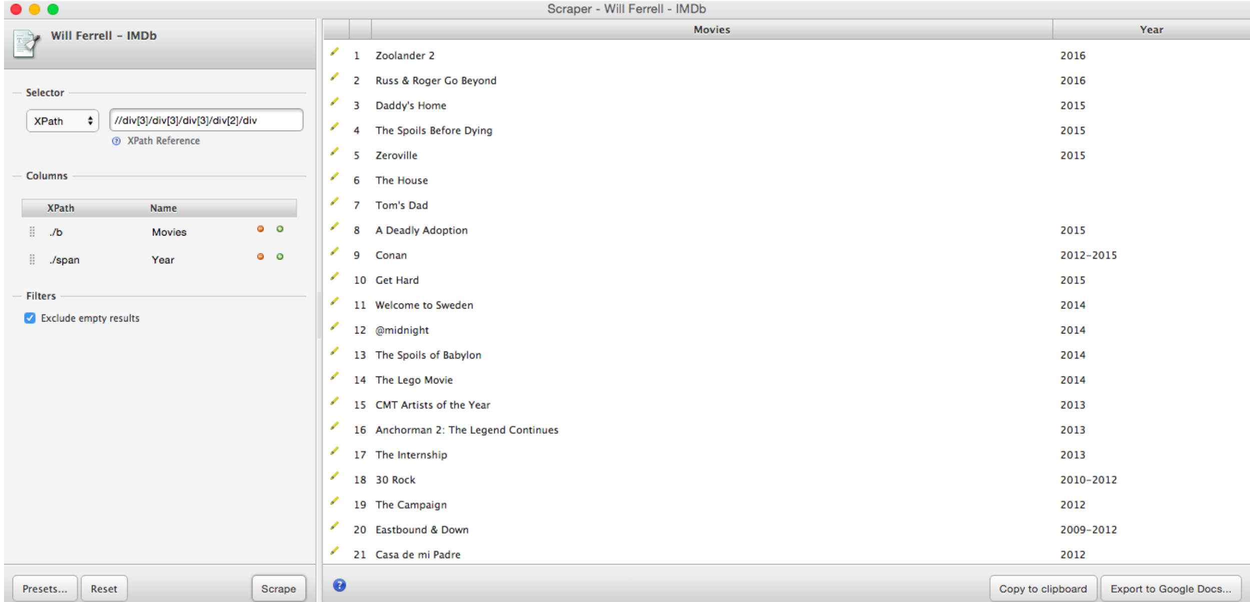


Fig. 7j - Final scraping result

Other tools like Kimono labs and import.io are also very easy to use and can be used for reliable web extraction.

Chapter 8

HTTP Retrieval with HTML Parsing

Most of the time web scraping toolkits do not produce desirable results. Most data scientist need more flexibility from a web scraper than what is typically provided in a web scraping toolkit. Because of this, data scientists prefer to create their own scraper.

Scrapers usually have three steps:

- Manipulating URLs
- Scraping using XPath
- Looping

Before diving into URL manipulation and XPath, it's essential that HTML fundamentals are covered.

HTML

HTML Hyper Text Markup Language is the standard markup language used to create web pages. The basic syntax for writing HTML is to write HTML elements consisting of tags enclosed in angle brackets. Each HTML tag has it own definition. A limited number of HTML tags and their definitions are listed in the image below.

Tag	Description
<html> ... </html>	Declares the Web page to be written in HTML
<head> ... </head>	Delimits the page's head
<title> ... </title>	Defines the title (not displayed on the page)
<body> ... </body>	Delimits the page's body
<h *n*> ... </h*n*>	Delimits a level *n* heading
 ... 	Set ... in boldface
<i> ... </i>	Set ... in italics
<center> ... </center>	Center ... on the page horizontally
...	Brackets an unordered (bulleted) list
...	Brackets a numbered list
...	Brackets an item in an ordered or numbered list
 	Forces a line break here
<p>	Starts a paragraph
<hr>	Inserts a horizontal rule
<img src="...">	Displays an image here
 ... 	Defines a hyperlink

Fig. 8a - HTML Tags

It is important to know the definition of HTML tags as the information on any web page is wrapped around different HTML tags. To scrape any piece of information from a web page, there is a need to know the unique set of HTML tags governing that information.

To scrape data successfully from any web page, one doesn't have to be an HTML expert, one needs only to know the HTML structure and HTML tags.

Manipulating URLs

GET,PUT and POST response

A GET request requests data from a specified source whereas POST request submits data to be processed to a specified resource. A GET request sends the query string (name/value pair) in the URL. For example, in the following www.foo.com/blah.do?stuff=other, the string after the question mark '?' contains the request parameter for the name/value where the name is stuff and the value is other. The following

The following is a put request on the Google resource.

https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=put%20request

Changing the URL q=get change the google search from a put request to a get request.

https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=get%20request

A POST request sends the query string (name/value pair) in the message body.

Column1	GET	POST
BACK button/Reload	Harmless	Data will be re-submitted (the browser should alert the user that the data are about to be re-submitted)
Cached	Can be cached	Not cached
Bookmarked	Can be bookmarked	Cannot be bookmarked
Encoding type	application/x-www-form-url encoded	application/x-www-form-urlencoded or multipart/form-data. Use multipart encoding for binary data
Restrictions on data length	Yes, when sending data, the GET method adds the data to the URL; and the length of a URL is limited (maximum URL length is 2048 characters)	No restrictions
Restrictions on data type	Only ASCII characters allowed	No restrictions. Binary data is also allowed
History	Parameters remain in browser history	Parameters are not saved in browser history
Security	GET is less secure compared to POST because data sent is part of the URL. Never use GET when sending passwords or other sensitive information!	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

Fig. 8b - GET vs POST

A good scraper should have a parameterized GET URL such that any user can change the URL to scrape similar pages. For example in the R code below we want to scrape the 2014-2015 season statistics for Boston Bruins entire team.

```
1  > var<-8470627
2  > print(paste("http://bruins.nhl.com/club/player.htm?id=",var))
3  [1] "http://bruins.nhl.com/club/player.htm?id= 8470627"
4  #note the space in between URL
```

The paste() function can be used to manipulate the URL but by default the separator is assumed by the paste function to be a single space; however spaces are not allowed in the name of a URL. To change the default argument pass the sep argument to the paste function with the desired empty character “”.

```
1  > var<-8470627
2  > url<-paste("http://bruins.nhl.com/club/player.htm?id=",var,sep="")
3  > url
4  [1] "http://bruins.nhl.com/club/player.htm?id=8470627"
5  > browseURL(url)
6  #No space in URL and hence can be opened in browser
```

The above example provides the URL for just one player but we can increment the value of the var variable to fetch a different URL or a specific player’s statistics can be fetched by manipulating the var variable.

```
1 > var<-8470627
2 > for(i in 1:3){
3 + print(paste("http://bruins.nhl.com/club/player.htm?id=",var,sep=""))
4 + var<- var+i
5 + }
6 [1] "http://bruins.nhl.com/club/player.htm?id=8470627"
7 [1] "http://bruins.nhl.com/club/player.htm?id=8470628"
8 [1] "http://bruins.nhl.com/club/player.htm?id=8470630"
```

Now all the above URLs can be passed as an argument to the actual web scraping function.

Web scraping packages

- RCurl - The RCurl package is an R-interface to the libcurl library that provides HTTP facilities. This allows us to download files from Web servers by getting forms. The primary top-level entry points are : getURL(), getURLContent()
- XML - The XML package is necessary to parse the XML and HTML code. This also offers access to an XPath interpreter.
- scrapeR - The scrapeR package is necessary to extract the data from the XML and HTML documents. It provides a function scrape() that assists the user with retrieving HTML and XML files, parsing their contents and diagnosing potential errors that may occur along the way.

Case study of actual scraping

Let's say that we need historical tax information for properties in Boston. This data is available through the web at www.cityofboston.gov, although the city does not provide the data for download.

We will build an R script that "scrapes" the needed data from the relevant web page on the website for the desired property.

In order to explain web scraping, we will consider the example of scraping the following website to extract some useful data on Northeastern University. Website to be scraped:

http://www.cityofboston.gov/assessing/search/?pid=0402236000[11]

[11]http://www.cityofboston.gov/assessing/search/?pid=0402236000

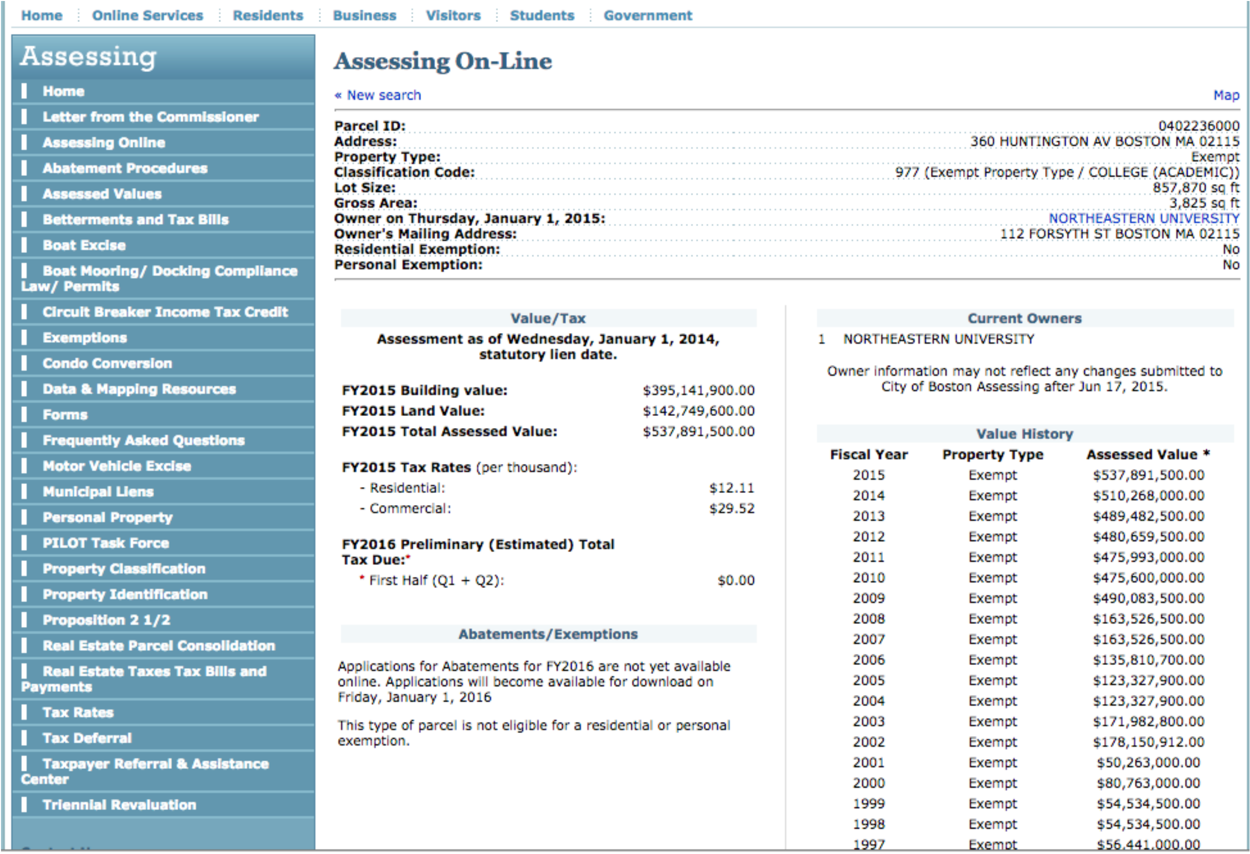


Fig. 8c - website to be scraped

Before going to the actual scraping exercise let's look at some sample code that accesses information on this web page.

> *NOTE* - Source code of any web page can be easily accessed depending on the browser used, for example to access the source code of any webpage on Google chrome just right click anywhere on the page and select inspect element. It is advisable to select the data of your interest and then click on inspect element so that code is directly displayed on the highlighted part of the page.

Let's start by building a getUrl function which will take the pid as an argument and return the URL from which we need to scrape the data.

```
1  > getUrl<- function (pid){
2  +   URL<-paste("http://www.cityofboston.gov/assessing/search/?pid=",pid,sep="")
3  +   return(URL)
4  + }
5
6  > pid<-"0402236000"
7  > url<-getUrl(pid)
8
9  > browseURL(url)
```

> *NOTE* - Do not create a function named getURL, since you will want to use the getURL built-in function in the RCurl library. If you do create your own getURL function, R will overload the getURL function and your newly created function will be the default getURL function. To specify the getURL function in RCurl, use its full name RCurl::getURL.

```
1  > getURL<- function (pid){
2  + URL<-paste("http://www.cityofboston.gov/assessing/search/?pid=",pid,sep="")
3  + return(URL)
4  + }
5  > pid<-"0402236000"
6  > url<-getURL(pid)
7
8  > webpage <- getURL(url)
9  > webpage
10 [1] "http://www.cityofboston.gov/assessing/search/?pid=http://www.cityofboston.g\
11 ov/assessing/search/?pid=0402236000"
12
13 > webpage <- RCurl::getURL(url) #global scope
14 [1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">\r\n<html>\
15 \r\n<head>\r\n<title>Parcel 0402236000 - City of Boston</title>\r\n <meta name=\\
16 "keywords\" content=\"Boston\" />\r\n <meta http-equiv=\"Content-Type\" content=\
17 \"text/html; charset=utf-8\" />\r\n \r\n <script type=\"text/javascript\" src=\"\
18 //m.cityofboston.gov/mobify/redirect.js\"></script>.............
```

So now we have the URL to scrape, the next step is that we need to pass the HTML to R and read it line by line. The RCurl::getURL() function is used for this purpose.

```
> library(RCurl)
> library(XML)
> webpage <- getURL(url)
> tc <- textConnection(webpage)
> webpage <- readLines(tc)
> close(tc)
> head(webpage,5)
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">"
[2] "<html>"
[3] "<head>"
[4] "<title>Parcel 0402236000 - City of Boston</title>"
[5] " <meta name=\"keywords\" content=\"Boston\" />"
```

Once we have a variable with the entire HTML code, we can extract the data enclosed within specific tags using the xpathApply() function.

XPath:

XPath (XML path language) is a query language for XML and HTML. It is used to traverse the HTML structure as well as select a particular node in the HTML tree. Once a node is selected, the function xmlValue extracts the value associated with that node. The specific path can be achieved by defining the CSS attributes combining with class and id of HTML tags.

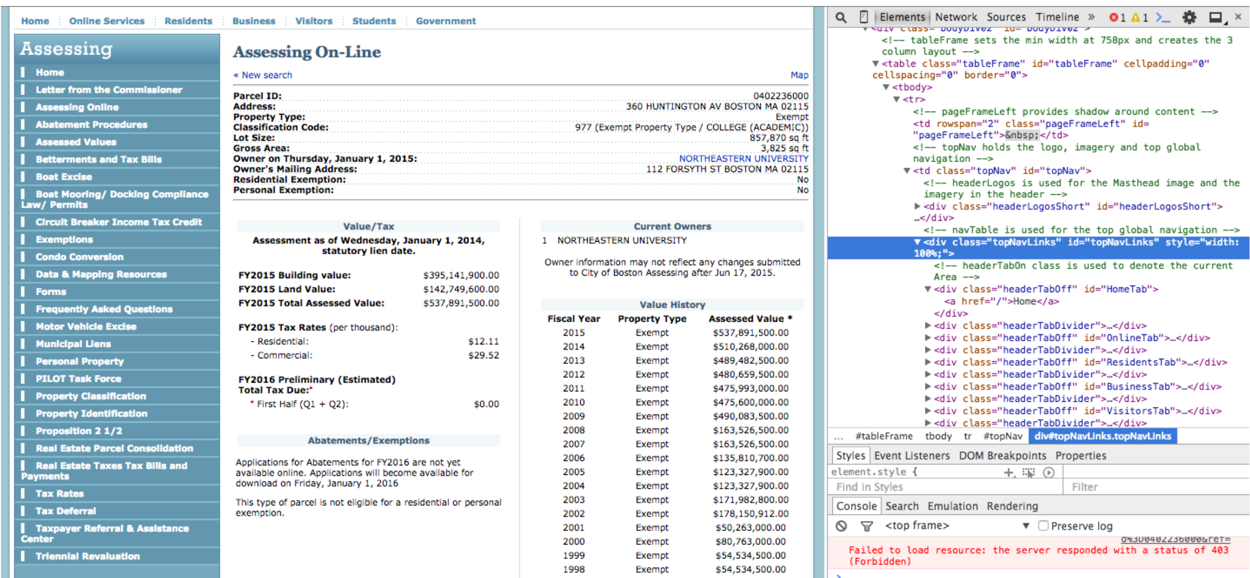


Fig. 8d - Highlighted code for top navigation bar

Fig. 8d shows the structure of the page to be scraped along with the HTML code.

```
1  > library(RCurl)
2
3  > library(XML)
4  > webpage <- getURL(url)
5  > tc <- textConnection(webpage)
6  > webpage <- readLines(tc)
7  > close(tc)
8
9  #useInternalNodes gives webpage a list context in r
10 > pagetree <- htmlTreeParse(webpage, useInternalNodes = TRUE)
11
12 #observe the xpathapply syntax and read explanation in text below
13 > header<- unlist(xpathApply(pagetree,"//*/div[@class='topNavLinks']/div[@class=\
14 'headerTabOff']/a",xmlValue))
15 > header
16 [1] "Home"             "Online Services" "Residents"       "Business"
17 [5] "Visitors"         "Students"        "Government"
```

The htmlTreeParse function in the XML package generates an R structure representing the XM-L//HTML tree.

In the example above the xpathApply statement is accessing the value associated with the top navigation bar. In the above statement, we passed three arguments:

1. Variable containing R structured HTML tree.
2. Pattern to find the appropriate node.
3. xmlValue to extract the value of the selected nodes.

Xpath syntax:

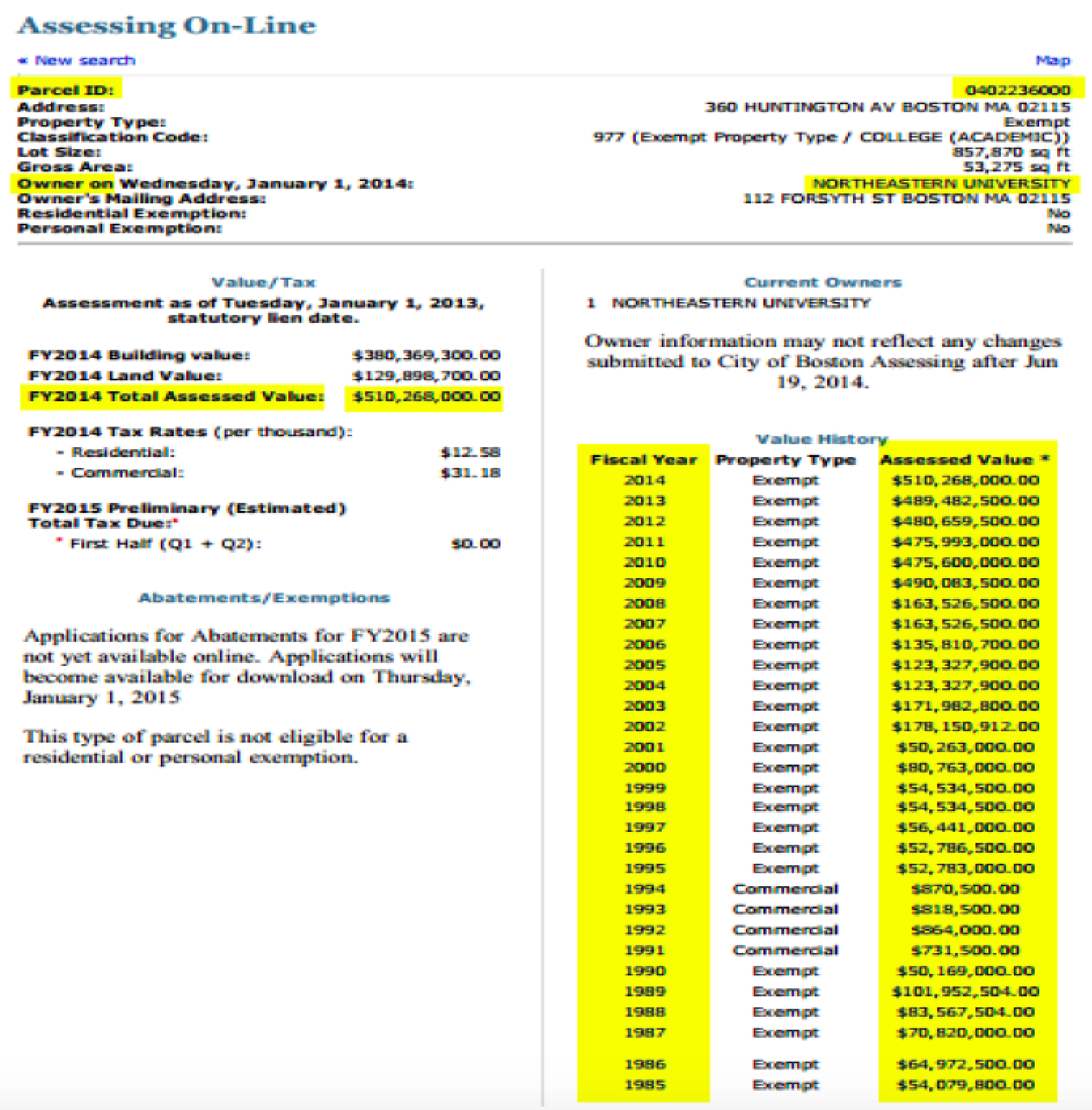
The second argument, which takes a pattern to find the appropriate node, works on the fundamental syntax of XPath:

1. A single slash (/) specifies to select a node from the root.
2. A double (slash (//) specifies to select nodes from anywhere in the document.
3. The ampersand (@) specifies to select CSS/HTML attributes enclosed within square brackets.

Just with these above three statements, any node can be accessed on any page with the correct pattern. In the example above, the XPath statement will read something like this: “Select all (*) nodes anywhere in the document with div class = topNavlinks, div class = headerTabOff and link (a)”

Actual scraping example

All the information on this page is not desirable, so we will first define what all information is needed for our purpose and then we will scrape that information.



Assessing On-Line

« New search Map

Parcel ID:	0402236000
Address:	360 HUNTINGTON AV BOSTON MA 02115
Property Type:	Exempt
Classification Code:	977 (Exempt Property Type / COLLEGE (ACADEMIC))
Lot Size:	857,870 sq ft
Gross Area:	53,275 sq ft
Owner on Wednesday, January 1, 2014:	NORTHEASTERN UNIVERSITY
Owner's Mailing Address:	112 FORSYTH ST BOSTON MA 02115
Residential Exemption:	No
Personal Exemption:	No

Value/Tax

Assessment as of Tuesday, January 1, 2013, statutory lien date.

FY2014 Building value:	$380,369,300.00
FY2014 Land Value:	$129,898,700.00
FY2014 Total Assessed Value:	$510,268,000.00

FY2014 Tax Rates (per thousand):

- Residential:	$12.58
- Commercial:	$31.18

FY2015 Preliminary (Estimated) Total Tax Due:*

* First Half (Q1 + Q2):	$0.00

Abatements/Exemptions

Applications for Abatements for FY2015 are not yet available online. Applications will become available for download on Thursday, January 1, 2015

This type of parcel is not eligible for a residential or personal exemption.

Current Owners

1 NORTHEASTERN UNIVERSITY

Owner information may not reflect any changes submitted to City of Boston Assessing after Jun 19, 2014.

Value History

Fiscal Year	Property Type	Assessed Value *
2014	Exempt	$510,268,000.00
2013	Exempt	$489,482,500.00
2012	Exempt	$480,659,500.00
2011	Exempt	$475,993,000.00
2010	Exempt	$475,600,000.00
2009	Exempt	$490,083,500.00
2008	Exempt	$163,526,500.00
2007	Exempt	$163,526,500.00
2006	Exempt	$135,810,700.00
2005	Exempt	$123,327,900.00
2004	Exempt	$123,327,900.00
2003	Exempt	$171,982,800.00
2002	Exempt	$178,150,912.00
2001	Exempt	$50,263,000.00
2000	Exempt	$80,763,000.00
1999	Exempt	$54,534,500.00
1998	Exempt	$54,534,500.00
1997	Exempt	$56,441,000.00
1996	Exempt	$52,786,500.00
1995	Exempt	$52,783,000.00
1994	Commercial	$870,500.00
1993	Commercial	$818,500.00
1992	Commercial	$864,000.00
1991	Commercial	$731,500.00
1990	Exempt	$50,169,000.00
1989	Exempt	$101,952,504.00
1988	Exempt	$83,567,504.00
1987	Exempt	$70,820,000.00
1986	Exempt	$64,972,500.00
1985	Exempt	$54,079,800.00

Fig 8e. - Highlighted information will be scraped

As we can see in the above image, only the information highlighted in yellow is extracted.

```
# Load the required libraries
library(RCurl)
library(XML)

#Set the working directory to your workspace
setwd("C:/Users/Martin/Desktop")

# This is the URL of the website we need scrape to get information on the
# total assessed value of Northeastern University's properties

getUrl<- function (pid){

   URL<-paste("http://www.cityofboston.gov/assessing/search/?pid=",pid,sep="")

   return(URL)
}
 pid<-"0402236000"

 #This pid is the value of pid given in URL of webpage

theurl<-getUrl(pid)

webpage <- getURL(theurl)

# convert the page into a line-by-line format rather than a single string
tc <- textConnection(webpage)
webpage <- readLines(tc)
close(tc)


pagetree <- htmlTreeParse(webpage, useInternalNodes = TRUE)

parcelID <- unlist(xpathApply(pagetree,"//*/table[@width='100%']
             [@cellpadding='0']/tr[3]/td",xmlValue))
parcelID

ownerName <- unlist(xpathApply(pagetree,"//*/table[@width='100%']
             [@cellpadding='0']/tr[9]/td",xmlValue))
ownerName

totalValueLabel <- unlist(xpathApply(pagetree,"//*/table[@width='100%']
             /tr[5]/td/b",xmlValue))
```

```
43
44 ttLabel<- unlist(strsplit(totalValueLabel, ":"))
45 ttLabel[2]
46
47 totalValueText <- unlist(xpathApply(pagetree,"//*/table[@width='100%']
48                 /tr[5]/td[@align = 'right']",xmlValue))
49
50 ttText <- unlist(strsplit(totalValueText, " "))
51
52 ttText <- gsub(pattern = "([\t\n])",
53                    replacement ="" , x = ttText[2], ignore.case = TRUE,
54                    perl = FALSE, fixed = FALSE, useBytes = FALSE)
55 ttText
56
57 x <- unlist(xpathApply(pagetree, "//*/table[@width='100%']/tr[2]
58                 /th[@align='center']", xmlValue))
59 y <- unlist(xpathApply(pagetree, "//*/table[@width='100%']/tr
60                 /td[@align='center']", xmlValue))
61 content <- as.data.frame(matrix(y, ncol = 3, byrow = TRUE))
62 new.line.3 <- gsub(pattern = "([\t\n])",
63                    replacement ="" , x = x, ignore.case = TRUE,
64                    perl = FALSE, fixed = FALSE, useBytes = FALSE)
65 new.line.3
66 content
```

The above code can be broken down into 5 major steps which will serve as a good template to make a scraper from scratch.

1. URL manipulation
2. Get the web page via its URL
3. Parse the HTML that defines the page
4. Extract leaf items which have the data
5. Clean the extracted data

URL manipulation -

Making a function to generate an URL makes the scraper parameterized and easy to use.

```
getUrl<- function (pid){

  URL<-paste("http://www.cityofboston.gov/assessing/search/?pid=",pid,sep="")

  return(URL)
}
```

Get the web page via its URL

Download the raw HTML content of the webpage using these two functions:

```
webpage <- getURL(theurl)

webpage <-readLines(tc<-textConnection(webpage));
```

Parse the HTML that defines the page

Transform the raw HTML into a more convenient format to work with using htmlTreeParse(). Setting useInternalNodes=TRUE allows one to access the parent and ancestor nodes.

```
pagetree <- htmlTreeParse(webpage, useInternalNodes = TRUE)
```

Extract leaf items which have the data

Use xpathApply() to extract the leaf items in the HTML document. To eliminate undesired matches, the query restricts the high-level table attribute to width=100% and table heading attribute aligned to center. xmlValue is convenient for extracting the text value of the node.

```
parcelID <- unlist(xpathApply(pagetree,"//*/table[@width='100%'][@cellpadding='0\
']/tr[3]/td",xmlValue))
parcelID

ownerName <- unlist(xpathApply(pagetree,"//*/table[@width='100%'][@cellpadding='\
0']/tr[9]/td",xmlValue))
ownerName

totalValueLabel <- unlist(xpathApply(pagetree,"//*/table[@width='100%']/tr[5]/td\
/b",xmlValue))
ttLabel<- unlist(strsplit(totalValueLabel, ":"))
ttLabel[2]

totalValueText <- unlist(xpathApply(pagetree,"//*/table[@width='100%']/tr[5]/td[\
@align = 'right']",xmlValue))
```

The most important argument for xpathApply is to recognize the unique pattern required to access a particular node, CSS/HTML attributes are used to build a unique pattern. Any pattern can be made more specific by adding more attributes. for example, in an earlier example to fetch the header tab, we can make the xpath more specific by adding the id attribute (@id='Home Tab').

```
1 > header<- unlist(xpathApply(pagetree,"//*/div[@class='topNavLinks']/div[@class=\
2 'headerTabOff']/a",xmlValue))
3 > header
4 [1] "Home"            "Online Services" "Residents"       "Business"
5 [5] "Visitors"        "Students"        "Government"
6 > home<- unlist(xpathApply(pagetree,"//*/div[@class='topNavLinks']/div[@id='Home\
7 Tab']/a",xmlValue))
8 > home
9 [1] "Home"
```

Clean the extracted data

The R global substitution function gsub() changes the "\t\n" combination to an empty string(""). Cleaning scraped data is another task imprtant task for the data scientist.

```
1 new.line.3 <- gsub(pattern = "([\t\n])",
2                    replacement ="" , x = x, ignore.case = TRUE,
3                    perl = FALSE, fixed = FALSE, useBytes = FALSE)
```

Finally, data is scraped, cleaned and ready for the analysis.

```
> new.line.3
[1] "Fiscal Year"      "Property Type"     "Assessed Value *"
> content
     V1         V2              V3
1  2015     Exempt $537,891,500.00
2  2014     Exempt $510,268,000.00
3  2013     Exempt $489,482,500.00
4  2012     Exempt $480,659,500.00
5  2011     Exempt $475,993,000.00
6  2010     Exempt $475,600,000.00
7  2009     Exempt $490,083,500.00
8  2008     Exempt $163,526,500.00
9  2007     Exempt $163,526,500.00
10 2006     Exempt $135,810,700.00
11 2005     Exempt $123,327,900.00
12 2004     Exempt $123,327,900.00
13 2003     Exempt $171,982,800.00
14 2002     Exempt $178,150,912.00
15 2001     Exempt  $50,263,000.00
16 2000     Exempt  $80,763,000.00
17 1999     Exempt  $54,534,500.00
18 1998     Exempt  $54,534,500.00
19 1997     Exempt  $56,441,000.00
20 1996     Exempt  $52,786,500.00
21 1995     Exempt  $52,783,000.00
22 1994 Commercial     $870,500.00
23 1993 Commercial     $818,500.00
24 1992 Commercial     $864,000.00
25 1991 Commercial     $731,500.00
26 1990     Exempt  $50,169,000.00
```

Fig. 8f - Cleaned data

Chapter 9

Data collection through web APIs

As the internet continues to grow, the amount of data available has increased substantially. Web API's make data available through a request-response interface similar to a function call.

A web API is an application programming interface (API) for either a web server or a web browser called server side API and client side API respectively. Client side web API extends the browser functionality via standardized javascript. Server side web API defines a request-response message system using either JSON or XML. JSON or XML are used to encode request and response.

In other words, API is an Application Programming Interface that provides call functionality of some service. The service can be computational or data related (storage/retrieval). Web APIs are different than traditional web services that are based on SOAP; they are more light-weight as they use the REST architecture.

REST

REST stands for REpresentational State Transfer. REST is an architectural style, a communication protocol to connect between machines using simple HTTP to make calls. RESTful applications uses HTTP for all four CRUD operations (Create/Read/Update/Delete). Rest services are:

1. Platform independent
2. Language independent
3. Use Standardized protocols

Let's look at a very simple example of a query using API through R.

```
1  Query='select share_count,like_count,comment_count from link_stat where url="'
2  pageUrl="http://bigthink.com/praxis/a-three-question-quiz
3                                  -to-test-your-rationality"
4
5  APIUrl = paste('http://graph.facebook.com/fql?q=',Query,pageUrl,'"')
6  lookUp <- URLencode(APIUrl)
7  browseURL(lookUp)
```

The above code uses a query statement to retrieve the number of times a specific web page is shared on Facebook, liked on Facebook and the number of comments.

> *NOTE* - the URLencode function used in the above code, removes spaces from the URL name and replaces them with the hexadecimal value for a space i.e. "%20".

Structure of web API requests

1. API endpoint using URL manipulation.
2. Retrieving data generated by API in either JSON or XML format.
3. Parametrizing the requests.

Let's see how we can use this structure with the above API query.

```
1  > library(rjson)
2  > library(RCurl)
3
4  > fbStats<-function(URL){
5  +   Query='select share_count,like_count,comment_count
6                                   from link_stat where url="'
7  +   APIUrl = paste('http://graph.facebook.com/fql?q=',Query,URL,'"')
8  +   APIUrl <- URLencode(APIUrl)
9  +   r<-getURL(APIUrl) #same code as web scraping
10 +   root<-fromJSON(r) #converting JSON to list like we converted HTML
11 +   share<-root$data[[1]]$share_count
12 +   likes<-root$data[[1]]$like_count
13 +   comments<-root$data[[1]]$comment_count
14 +   #extracted data from the list
15 +   return(data.frame(share,likes,comments))
16 + }
17
18 > URL="http://bigthink.com/praxis/a-three-question-quiz-to-test-your-rationality"
19 # Instead of above link you can use the link of any page shared on Facebook
20 > stats<-fbStats(URL)
21 > stats
22   share likes comments
23 1  1775  4266     1462
```

This is the structured code for the same API request, let's see the structure in detail:

API endpoint using URL manipulation

URL manipulation is used to reach the API endpoint, where the response from the API is displayed. If you browse the content of APIUrl variable, without the quotes around it, you can observe the response from API.

```
{
   "data": [
      {
         "share_count": 1775,
         "like_count": 4266,
         "comment_count": 1462
      }
   ]
}
```


Fig. 9a - Response from API

Details of endpoint URL of famous API's can be found on this link of compiled API list. API directory[12]

Retrieving data generated by API in either JSON or XML format-

After reaching the API endpoint in a correct way, the next step requires to fetch the data and shape for analysis. Data is retrieved differently depending upon the format of the response.

a. XML data - Use library XML and functions xmlTreeParse() or xmlToDataFrame() to pass the data to R and then manipulate data in R.
b. JSON data - JSON data first needs to be passed to R, we can either use readLines() function or the getURL() function from the RCurl library as both the functions are familiar.

NOTE - The readLines function cannot parse a secure web page i.e. if the API endpoint is returning a secure page (HTTPS://…) then use the getURL function.

Once we have the data in R, we use the fromJSON() function within the rjson library to convert the format into a JSON list which can be easily manipulated in R using $ sign to capture the recurring instance of anything in the JSON list.

```
1  > library(rjson)
2  > library(RCurl)
3
4  > URL="http://bigthink.com/praxis/a-three-question-quiz-to-test-your-rationality"
5  > Query='select share_count,like_count,comment_count from link_stat where url="'
6  > APIUrl = paste('http://graph.facebook.com/fql?q=',Query,URL,'"')
7  > APIUrl <- URLencode(APIUrl)
8  > r<-getURL(APIUrl)
9  > root<-fromJSON(r)
10
```

[12] http://www.programmableweb.com/apis/directory

```
> root
$data
$data[[1]]
$data[[1]]$share_count
[1] 1775

$data[[1]]$like_count
[1] 4270

$data[[1]]$comment_count
[1] 1462
```

Parameterizing the requests

It's always a good idea to parameterize the request in order to increase the ease of usage of the API request. In the above example, we made a function called fbStats which takes a URL as an argument and returns the share counts, the likes counts and the comment counts of this URL on Facebook. By making it parameterized, we can now simply change the value of the URL variable and can fetch the Facebook statistics of any page.

This was just a simple example of API request processing, Next we will see nesting API requests using static maps API and finally we will see YouTube API and twitter API which require two different authentication process before actually fetching any data.

Nesting API requests

```
library(RCurl)
library(rjson)
library(plyr)

#getLocation function returns the latitude and longitude of
#any address via google maps api.

#fetchUrl function gives the url to fetch json string (endpoint url)
fetchUrl <- function(address) {
  root <- "http://maps.google.com/maps/api/geocode/"   #root url
  u <- paste(root, "json", "?address=", address, "&sensor=false", sep = "")
  return(URLencode(u)) #encoding the url
}


getLocation <- function(address) {
  #getting the url for the address,
```

```
18    #I suggest to take a look at this URL before further exploring the code
19    url <- fetchUrl(address)
20    #just run this fetchUrl command outside the function and open the URL in brows\
21  er
22
23    json <- getURL(url)
24    x <- fromJSON(json) #getting the json
25    if(x$status=="OK") {
26    #checking if the Address url used produces the correct json string
27      lat <- x$results[[1]]$geometry$location$lat
28      lng <- x$results[[1]]$geometry$location$lng
29      location_type <- x$results[[1]]$geometry$location_type
30      formatted_address <- x$results[[1]]$formatted_address
31      return(c(lat, lng, location_type, formatted_address))
32      Sys.sleep(0.5)
33    } else {
34      return(c(NA,NA,NA,NA))
35    }
36  }
37  address <- c("Northeastern Universty, Boston, MA",
38                      "Harvard University, Boston, MA")
39  locations <- ldply(address, function(x) getLocation(x))
40
41  names(locations) <- c("lat", "lon", "location_type", "formatted")
42  locations
43
44  #Another function to access static maps
45  staticURL<-function(latitude,longitude,zoom,maptype){
46  base="http://maps.googleapis.com/maps/api/staticmap?center="
47  suffix ="&size=800x800&sensor=false&format=png"
48
49  target <- paste0(base,latitude,",",longitude,"&zoom=",zoom,
50                                      "&maptype=",maptype,suffix)
51
52  return(target)
53  }
54
55  test<-staticURL(locations$lat[1],locations$lon[1],"13","hybrid")
56  #enter latitude and longitude, zoom values range from 1 - 18,
57  #maptypes can be- hybrid,satellite,terrain,roadmap
58
59  browseURL(test)
```

```
60 download.file(test,"test.png", mode = "wb")
```

The above example is an example of nesting API requests, since we pass the result from one API to another API. Our example uses Google maps' geocoding API to fetch the longitude and latitude of a specific location. We then pass these returned results to Google's static maps API to retrieve a map file for this specific location.

Using APIs after authentication

The authentication process is supplied to make sure that data is secure and cannot be misused. By generating these keys, you are actually registering your credentials so that an API manager can track who is using what data. There are many types of authentication processes, but we will discuss only two types in this book.

1. Generating a consumer key and oauth token - Twitter API
2. Generating a browser key - YouTube API

Twitter API

We will use the Twitter Streaming API to get tweet data using R. Through the Twitter WebAPI, external applications can retrieve Twitter tweet data in JSON format. If a user wants to access the tweet data through the Web API, a user needs to register the application with Twitter.

Registering your Application on Twitter.

To run this script, you need to generate your own "consumerKey", "consumerSecret", oauth_token and oauth_token_secret by registering your application with Twitter. The process is the following:

1. Register it here: Twitter Apps[13]. Go to Create New App.
2. Enter a name for your application. e.g "Data Science NEU". Enter a description for the app e.g. "Getting tweet data using streaming API in R". For the website, enter a placeholder e.g. "http://www.google.com". The callback URL field is optional. Scroll down the page. Check the license agreement. And click on "Create your Twitter Application". Your application has been successfully created.
3. Under the API Keys Tab, you will see an API Key and an API Secret. For the oAuth_token and oauth_token_secret scroll down the page and click on the button "Create my access token". Refresh the page if necessary and you will now see the access token and the access token secret. Note: To check the current working directory use the command getwd() in RStudio.

Once you have the twitter credentials ready, enter the keys in the code given below:

[13] https://dev.twitter.com/apps

Environment Setup:

Step 1: Change the working directory from “C:/Users/loginName/Documents” to “C:/Users/loginName/Desktop” using the following line of code. setwd(“C:/Users/loginName/Desktop”)

Step 2: Load the required libraries needed to use twitter.

- library(base64enc)
- library(RCurl)
- library(ROAuth)
- library(streamR)
- library(twitteR)
- library(httr)

NOTE: If you generate a message similar to this: “ Error in library(RCurl) : there is no package called ‘RCurl’ “ that means the RCurl package has not been installed. To fix the error, in Rstudio’s console window type the following command:

```
install.packages("RCurl")
```

and hit Enter. The package RCurl will be installed. Repeat the procedure for all the libraries you need to load.

Step 3: Download the certificate needed for authentication. This creates a certificate file on the desktop since you set your desktop as R’s working directory.

```
download.file(url="http://curl.haxx.se/ca/cacert.pem",destfile="cacert.pem")
```

Step 4: Create a file to collect all the Twitter JSon data received from the API call.

```
outFile <- "tweets_sample.json"
```

Step 5: Set all the Configuration details to authorize your application to access Twitter data.

```
#Twitter configuration
requestURL <- "https://api.twitter.com/oauth/request_token"
accessURL <- "https://api.twitter.com/oauth/access_token"
authURL <- "https://api.twitter.com/oauth/authorize"
consumerKey <- "XXXX"
consumerSecret <- "XXXX"
oauth_token <- "XXXX"
oauth_token_secret <-"XXXX"
```

The requestURL, accessURL and authURL remain the same. For the consumerKey, consumerSecret, oauth_token, oauth_token_secret you need to create a developer's account on Twitter. The XXXX strings should be replaced with the corresponding values from the twitter API web page.

Source Code

```
############################################################
##          Big Data with Twitter
##  Collecting Twitter Data from Straming API
## Sample intro course material. Demonstrates how to collect
## some basic data from Twitter (though streaming API).
##
## To run this script, you need to generate your own
## "consumerKey", "consumerSecret",
## and an oAuth token by registering your application with Twitter.
## It is a simple process, just pick a name for your application.
##
## Register it here: https://dev.twitter.com/apps
##
## Documentation of Twitter Streaming API:
##  https://dev.twitter.com/docs/streaming-apis/streams/public
##  https://dev.twitter.com/docs/auth/authorizing-request
##  http://www.foundations-edge.com/blog/oauth_in_R.html
##  https://dev.twitter.com/docs/streaming-apis/processing
##
## Prepared by Christoph Riedl
##     D'Amore-McKim School of Business &
##          College of Computer and Information Science
##
##      web: http://www.christophriedl.net
############################################################

## Change working directory: create folder and adjust according to taste
```

```
28 setwd("C:/Users/Brinal/Desktop")
29
30 # Load required libraries
31 library(base64enc)
32 library(RCurl)
33 library(ROAuth)
34 library(streamR)
35 library(twitteR)
36 library(httr)
37
38 download.file(url="http://curl.haxx.se/ca/cacert.pem", destfile="cacert.pem")
39
40 # Configuration for twitter
41 outFile     <- "tweets_sample.json"
42
43 # Twitter configuration
44 requestURL <- "https://api.twitter.com/oauth/request_token"
45 accessURL  <- "https://api.twitter.com/oauth/access_token"
46 authURL <- "https://api.twitter.com/oauth/authorize"
47 consumerKey  <- "XXXX"
48 consumerSecret  <- "XXXX"
49 oauth_token   <- "XXXX"
50 oauth_token_secret <- "XXXX"
51
52 my_oauth <- OAuthFactory$new(   consumerKey=consumerKey,
53                                 consumerSecret=consumerSecret,
54                                 requestURL=requestURL,
55                                 accessURL=accessURL, authURL=authURL)
56
57 my_oauth$handshake(cainfo="cacert.pem")
58
59 ##Once executing the above code returns true.
60 ##You will be given a link to authorize your application to get twitter feeds.
61 ##Copy the link in your browser. Click on Authorize MyApplication.
62 ##You will receive a pin number.
63 ##Copy the pin number and paste it in the console.
64 ##Once your application has been authorized you need
65 ## to register your credentials.
66
67 setup_twitter_oauth(consumerKey, consumerSecret, oauth_token, oauth_token_secret)
68
69 # Press 1 to allow the file to access the credentials
```

```
70
71 ##Now start reading tweets
72 sampleStream( file=outFile, oauth=my_oauth )
73
74 ##Alternative: a little more advanced if you want to filter for things
75 follow   <- ""    # TwitterIDs (not screennames!) of peple to follow
76 track    <- "Boston,RedSoxs"  # Comma-separated list of words to filter for
77 location <- c(23.786699, 60.878590, 37.097000, 77.840813)  # Geolocation of twee\
78 ts to filter for (see documentation)
79
80 filterStream( file.name=outFile, follow=follow, track=track,
81                 locations=location, oauth=my_oauth, timeout=10800)
82
83 #This creates a file on the desktop tweets_sample.json in which the tweet data w\
84 ill be stored.
```

Note: If the above code is not working on your system then its because of your antivirus software blocking the port to access twitter API. You need to turn off your firewall in order to run this code.

YouTube API

We use the YouTube API to fetch data using R. Using the YouTube API, we can retrieve information like the number of videos uploaded on any channel, retrieve the number of video views, the number of video likes, or the number of video dislikes. Data retrieved can be shaped using R making it conducive to analysis. To fetch any data using the YouTube API, a user must register as a Google console developer. This allows Google to track the data retrieved and potentially charge a user for its services. In this demonstration, we will play with a smaller dataset, which is free for public use. Like the twitter API, Google also provides the option of oauth authentication but in this example we will see a way to bypass oauth authentication by generating a browser key.

Registering your Application at the Google developer console

To run this script you need to generate your own "browserKey" by creating a new project at the Google developer console web page and registering for a YouTube API.

1. Login with your gmail ID on: Google developer console[14]
2. Click on create a project to create a new project for the youtube API. Enter a name for your project like "Youtube data collection". Once the project is all set, click on the APIs and auth to expand the links in the dashboard on the left and click on APIs. Click on the first link under the youtube APIs section named YouTube Data API and then click on enable API to use YouTube Data API v3.

[14] https://console.developers.google.com/project

3. The next step is to generate a browser key; a browser key is required to fetch any data from the YouTube API. Click on the credentials link, under the APIs link, then the auth tab on the left side of the dashboard. Under the public API access section, click on "create a new key". Different keys can be generated based on how the data is fetched. We are going to fetch data through the browser so we generate a browser key. Click on browser key, you can add referrers if you are fetching data from any other website. In this example, we leave the field blank since we are retrieving the data from R. Click on create and copy the generated key for later use.

Environment Setup :

Step 1: Change the working directory to Desktop using setwd command:

Mac users: setwd("/Users/username/Desktop")

Windows: setwd("C:/Users/username/Desktop")

Step 2: Load the following libraries after installing their packages.

```
1 install.packages("RCurl")
2 install.packages("rjson")
3 library(RCurl)
4 library(rjson)
```

The RCurl library is used to scrape data from webpages. We will use the function getURL to retrieve the HTML elements into R variables. The rjson library provides functions to manipulate json elements. In this example, we use the fromJSON function to convert the JSON string to an R list.

Step 3: Set the browser key to the key generated in the above steps:

```
1 Key<- "XXXX"
```

Step 4: Obtaining a YouTube video id:

In this first example, we will fetch statistics on a video. This example, will ensure our API connection provides connectivity. To fetch a video id, Open YouTube and browse to any video that you would like to collect statistics. After you are on the video page, the URL will look some like this: https://www.youtube.com/watch?v=RgKAFK5djSk[15]

In this URL, anything after the "v=" is the video ID. We will need to pass this video ID as an argument to our stats function.

Step 5: Obtaining a YouTube channel id:

[15]https://www.youtube.com/watch?v=RgKAFK5djSk

As part of our second example, we will fetch statistics of videos on a particular YouTube channel. To get a channel ID go to YouTube[16] and on the left side dashboard click on browse channels. Using the search channel search box, search for any channel. Click on the link for any channel, the URL should look something like this: https://www.youtube.com/channel/UCz6NJuz0ss3TxW7Fw4h_KIg[17]

In this URL anything after channel, is a channel ID i.e. UCz6NJuz0ss3TxW7Fw4h_KIg.

> *NOTE* - sometimes you might get a URL like this: https://www.youtube.com/user/superherosachin[18] It just means that this is not a channel but a user. Click some other link to find the channel.

Source Code

```
1  ############################################################
2  ##         Big Data with YouTube
3  ##  Collecting YouTube Data from Streaming API
4  ##
5  ##
6  ##
7  ## To run this script, you need to generate your own "browserKey"
8  ## by registering your application with Google.
9  ## It is a simple process, just pick a name for your project.
10 ## Register it here: https://console.developers.google.com/project
11 ##
12 ##          Collecting Data via WebAPIs in R
13 ##          Martin Schedlbauer, Ph.D., Yatish Jain
14 ##
15 ## Prepared by  Yatish Jain
16 ##     email: jain.ya@husky.neu.edu
17 ##
18 ##
19 ##
20 ##
21 ############################################################
22
23 ## change the working directory to desktop to keep track of output file.
24 setwd("/Users/username/Desktop")
25 getwd()
26
27 # Load required libraries
```

[16] http://www.youtube.com
[17] https://www.youtube.com/channel/UCz6NJuz0ss3TxW7Fw4h_KIg
[18] https://www.youtube.com/user/superherosachin

```
28 library(rjson)
29 library(RCurl)
30
31 #Generate your own key as per the instructions in document
32 #and paste it here to configure the API
33 key<- "XXXX"
34
35
36 #Funtion to check connection.
37 #This getStats function will fetch the statistics of
38 #any video given the video ID and key.
39 #Read instructions on how to get the video ID under the environment setup section
40
41 getStats <- function(id,key){
42   url=paste("https://www.googleapis.com/youtube/v3/videos?id=",id,
43                         "&key=",key,"&part=statistics,snippet",sep="")
44   raw.data <- getURL(url)
45   rd  <- fromJSON(raw.data,unexpected.escape = "skip")
46   if(length(rd$items)!= 0){
47   title<- rd$items[[1]]$snippet$title
48   channelTitle<- rd$items[[1]]$snippet$channelTitle
49   description<-rd$items[[1]]$snippet$description
50   views<- rd$items[[1]]$statistics$viewCount
51   likes<- rd$items[[1]]$statistics$likeCount
52   if(is.null(likes)){
53     likes<-"Info Not available"
54   }
55   dislikes<- rd$items[[1]]$statistics$dislikeCount
56   if(is.null(dislikes)){
57     dislikes<-"Info Not available"
58   }
59   fav<- rd$items[[1]]$statistics$favoriteCount
60   if(is.null(fav)){
61     fav<-"Info Not available"
62   }
63   comments<- rd$items[[1]]$statistics$commentCount
64   if(is.null(comments)){
65     comments<-"Info Not available"
66   }
67
68   return(data.frame(title,description,channelTitle,views,likes,dislikes,fav,comm\
69 ents))
```

```
70     }
71   }
72 
73   #getVideos function return the list of videos along
74   #with their statistics given the channelID and key.
75   # Read the instructions on how to get channelID under the environment setup sect\
76   ion
77 
78   getVideos<- function(channelID,key){
79     url=paste("https://www.googleapis.com/youtube/v3/search?key=",key,
80       "&channelId=",channelID,"&part=snippet,id&order=date&maxResults=10",sep="")
81     raw.data <- getURL(url)
82     rd  <- fromJSON(raw.data)
83     perPage<- rd$pageInfo$resultsPerPage
84     totalResults<-rd$pageInfo$totalResults
85     totalVideos<-min(perPage,totalResults)
86   stats<-c(as.character(),as.character(),as.character(),as.integer(),
87                   as.integer(),as.integer(),as.integer(),as.integer())
88   for (i in 1:totalVideos){
89     kind<- rd$items[[i]]$id$kind
90     if(kind == "youtube#video"){
91       videoID<- rd$items[[i]]$id$videoId
92       print(videoID)
93       stats<-rbind(stats,getStats(videoID,key))
94 
95       }
96     else if(kind == "youtube#playlist"){
97       playlistID<- rd$items[[i]]$id$playlistId
98       url=paste("https://www.googleapis.com/youtube/v3/playlistItems?
99                 part=snippet%2CcontentDetails&maxResults=10&playlistId=",
100                                                    playlistID,"&key=",key,sep="")
101       raw.data <- getURL(url)
102       rd1  <- fromJSON(raw.data)
103       perPage<- rd1$pageInfo$resultsPerPage
104       totalResults<-rd1$pageInfo$totalResults
105       totalVideos<-min(perPage,totalResults)
106       for(i in 1:totalVideos){
107         videoID<-rd1$items[[i]]$contentDetails$videoId
108         print(videoID)
109         stats<-rbind(stats,getStats(videoID,key))
110       }
111     }
```

```
112
113 }
114
115 return(stats)
116 }
117
118
119 #getChannelsOrPlaylists function return the list of videos
120 #and their statistics associated with a keyword search on YouTube.
121 # When you search a keyword on youtube sometimes playlists
122 # also end up in search and we fetch the data from those
123 # playlists as well which might not be directly related
124 # to our search keyword, but it will fetch the data of similar searches.
125
126 getChannelsOrPlaylists<- function(search, key){
127   search<-URLencode(search)
128   url<-paste("https://www.googleapis.com/youtube/v3/search?q=",search,
129         "&key=",key,"&type=channel&part=snippet&maxResults=50",sep="")
130   raw.data <- getURL(url)
131   rd  <- fromJSON(raw.data)
132   perPage<- rd$pageInfo$resultsPerPage
133   totalResults<-rd$pageInfo$totalResults
134   totalChannels<- min(perPage,totalResults)
135   data<-c(as.character(),as.character(),as.character(),as.integer(),
136                   as.integer(),as.integer(),as.integer(),as.integer())
137   for(i in 1:totalChannels){
138     channelID<- rd$items[[i]]$id$channelId
139     print(channelID)
140     if(!is.null(channelID)){
141       data<-rbind(data,getVideos(channelID,key))
142     }
143   }
144   data<- data[complete.cases(data),]
145   data<-unique(data)
146   write.csv(data,"data.csv",row.names=F)
147
148 }
149
150 search<-"taylor swift"
151 getChannelsOrPlaylists(search,key)
```

Chapter 10

Storage of data

There are mainly two types of storage mechanisms:

- Relational Databases
- Non-relational Databases

Relational Databases

A relational database is a digital database used to store structured data with a given set of rules. A relational database is composed of relations or relational tables. A relation is manifested as a table. Data is stored in one or more tables, where the rows correspond to instances of the relation and the columns correspond to the variables or the attributes of the relation. In a relation, each row must be differentiated by a primary key. A key is a collection of fields that provide a unique value for each row.

When mapping the real world objects to tables, typically data is separated into different tables in order to minimize redundancy of data. Duplication of a particular data field within two tables, typically represents a relationship between the two tables. Before building a database, one needs to understand the types of questions that need to be answered from the data. The layout or schema of the database will be driven by this required functionality.

Relational Data Model

A relational data model represents all real world objects and relationships between objects as a two dimensional table. There are methods for representing a data model graphically, one such method is the Unified Modeling Language or UML. In UML, objects are represented as boxes and relationships between 2 objects are represented via lines. When a relationship exists among more than 2 objects, we represent the relationship as a diamond. In this chapter we consider only binary relationships; relationships between two objects.

For the purpose of this book, we will use a very small dataset of the library which can be accessed here[19].

This library dataset has the following columns: StudentID, StudentName, StudentEmail, BookName, BookGenre, AuthorName, AuthorDetails. Our goal is to convert this flattened version of the data into a collection of relations or tables. We need to identify the variables that are associated with real world objects. We do this by answering the following questions:

[19] https://drive.google.com/file/d/0B9uiGI8JEJw5X1U5dzNOdEhPcXc/view

1. What are the real world objects represented by these variables
2. What are the relationships between the real world objects?
3. What are the questions we want to answer from this data?

	A	B	C	D	E	F	G
1	StudentID	StudentNam	StudentEmail	BookName	BookGenre	AuthorName	AuthorDetails
2	1	Michelle	meecheli@gmail.com	And the mountains echoed	Historical fiction	Khaled Hosseini	Afghan-born American novelist and physician
3	2	Latika	latikas@yahoo.com	The bone clocks	Drama	David Mitchell	English novelist
4	3	Logan	log1123@icloud.com	Neuromancer	Science fiction	William gibson	American-Canadian speculative fiction novelist and essayist
5	4	Emma	emmij@gmail.com	A thousdand splendid suns	Novel	Khaled Hosseini	Afghan-born American novelist and physician
6	5	John	johnny@nytimes.com	Cloud atlas	Science fiction	David Mitchell	English novelist
7	6	Fred	freddie@tiny.cc	The peripheral	Science fiction	William gibson	American-Canadian speculative fiction novelist and essayist
8	7	Carl	carllie@soundcloud.com	The book theif	Young-adult fiction	Markus Zusak	¾Australian writer
9	8	Scott	rmedinaf@clickbank.net	And the mountains echoed	Historical fiction	Khaled Hosseini	Afghan-born American novelist and physician
10	9	Tammy	jbrooksc@eventbrite.com	The pillars of earth	Historical fiction	Ken follett	Welsh author of thrillers and historical novels
11	10	Wanda	ataylorh@xinhuanet.com	Kite runner	Fiction	Khaled Hosseini	Afghan-born American novelist and physician

Fig. 10a - Library dataset

We can insert this csv file directly into an SQL database but there is a problem with this if we want to minimize data redundancy. Let's look at the first 5 columns in the file, many students are issued the same book. For example Michelle and Scott are both issued the book "And the Mountains Echoed". We should not store all of the books data with each student. This is leading to redundancy of all book data fields whenever a student is assigned that specific book. Also, remember our goal is to identify a subset of columns that can be used as a primary key. We need to separate the real world student objects from the real world book objects. The way we do this is by creating a Student table and a Book table. We do not want to lose the Student's book assignment, so we create another table or field that allows us to store this relationship between specific student instances and book instances.

	A	B	C
1	ID	BookName	BookGenre
2	101	A Short History of Nearly Everything	Non-fiction
3	102	A thousdand splendid suns	Novel
4	103	A walk in the woods	Non-fiction
5	104	All quiet on the western front	War novel
6	105	And the mountains echoed	Historical fiction
7	106	Best war ever	Non-fiction
8	107	Blink	Non-fiction
9	108	Cloud atlas	Science fiction
10	109	Database System Concepts	Database
11	110	Down under	Travel literature

Fig. 10b - Book table

To create the Book table, we created a unique ID, that differentiates each row in the book table. This is the primary key of the book table. We also create a Student table that stores all of the known variables about the student. Once again we create a unique field for the student table, it is called StudentID. It is the primary key for the student table. In order to not lose the book assignment for each student, we store the BookID within the Student table to represent the assignment. We use the primary key,

BookID, from the Book table since this column is guaranteed to differentiate the different books. The BookId is found within both the Book Table and the Student Table, so there is some redundancy within the relational model. However, it is minimal redundancy since this redundancy represents the book assignment for the student. The other fields associated with the Book is not duplicated for each Student. When we have a field that represents a relationship between two tables, it is known as a foreign key. It is a field that represents a unique entity in another table. As expected, foreign keys need not be unique like primary keys. In our example, many students both Michelle and Scott are assigned to book 105. This does not violate any constraints on the schema.

	A	B	C	D
1	StudentID	StudentNam	StudentEmail	BookID
2	1	Michelle	meecheli@gmail.com	105
3	2	Latika	latikas@yahoo.com	119
4	3	Logan	log1123@icloud.com	113
5	4	Emma	emmij@gmail.com	102
6	5	John	johnny@nytimes.com	108
7	6	Fred	freddie@tiny.cc	123
8	7	Carl	carllie@soundcloud.com	120
9	8	Scott	rmedinaf@clickbank.net	105
10	9	Tammy	jbrooksc@eventbrite.com	124
11	10	Wanda	ataylorh@xinhuanet.com	112

Fig. 10c - Student table

Similarly, we can make a separate table for the real world authors that write books. This would remove the redundancy of the author's columns within the Book table. This can be done by making a foreign ID of AuthorID in Book table.

Before we move on to define the relationships between these tables to make a data model, let's define some keywords we will be using frequently.

Important Definitions

1. **Primary Key** is a field or collection of fields defined in a table to uniquely identify each row of a table. The primary key should be unique and not null for any row. The primary key is the selected candidate key to be used as the primary key.
2. **Foreign Key** is a field that references a unique row of another table. Unlike a primary key foreign key need not be unique.
3. **Attributes** are all the fields or columns of any table. For example BookID, BookName and BookGenre are the attributes of Book table.
4. **Entity** is a person, place, thing or concept about which data can be collected. Example Student, Book, Author are all entities. They correspond to real world objects.
5. **Composite Key** takes more than one attribute to uniquely identify an entity occurrence. When there is not one field that can differentiate the entity occurrences, we can use multiple columns to define the primary key.

6. **Candidate Key** is any field or collection of fields that qualify as a primary key. Example BookName can be a candidate key in book table.

Relationships

Within a relationship, we track the number of instances from each entity participating in the relationship. We also track if each instance in the table is required to participate in the relationship. If each instance in the relation is required to participate in the relationship, then the relationship is mandatory. If an instance of a relation is not required to participate in a relationship then the relationship is considered optional.

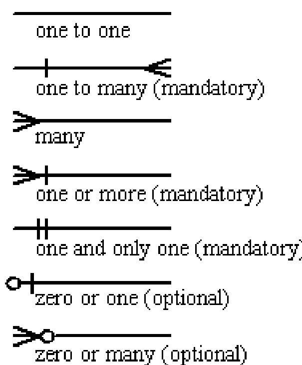


Fig. 10d - Relationship symbols

One-to-one relationship:

This relationship is rarely used in practice and is mainly used to segregate the set of columns which are rarely queried. When for each row of one table, there is a corresponding entry in another table, then the relationship between two tables is said to be a one to one relationship. For example: if we assume we will not be querying AuthorDetails much, then we can separate AuthorDetails column into a separate column.

One-to-many relationship:

A one to many relationship exists when for each row within one table there exists at least one corresponding row in the other table and at most many rows in the other table. In our example, one book can be issued to many students and one author can write more than one book.

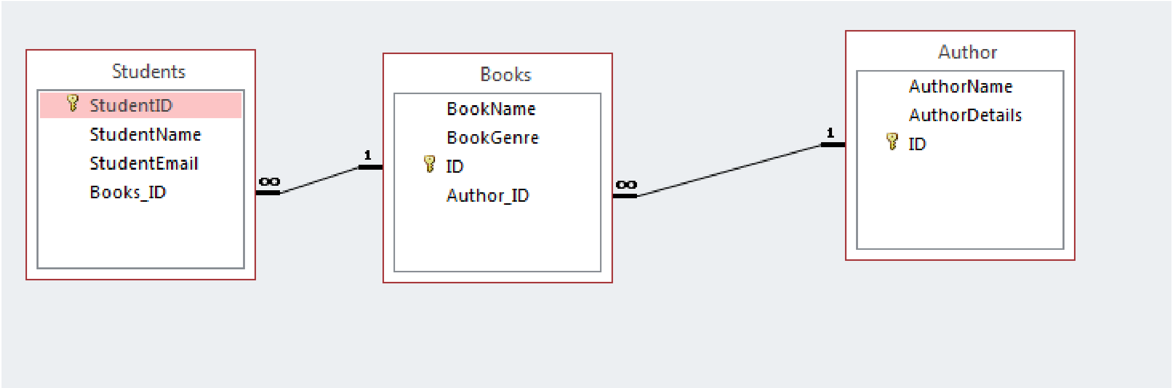


Fig. 10e - One to many relationship

NOTE - Check the other notation to mark the one to many relationship using the 1 and infinity symbols similar to single pipe and three arrows as shown in Fig. 10d.

Many-to-many relationship:

A many to many relationship exists when one or more rows of a table can be related to zero, one or more rows of another table. In our example we are working with a small dataset where one book has only one author but if we extend the schema to support multiple authors for a book, then we create a many to many relationship between Book and Author since 1 book can be written by many authors and 1 author can write many books.

Our current schema cannot support the many to many relationship, we need to create a table that allows us to store the corresponding Authors to a specific book. We create a table BooktoAuthor (intermediary) table that allows us to track the corresponding relationship between Authors and Books. The BooktoAuthor (intermediary) table would have a composite primary key that consists of the BookId and the AuthorId. Each row in this table represents a relationship between Books and Authors.

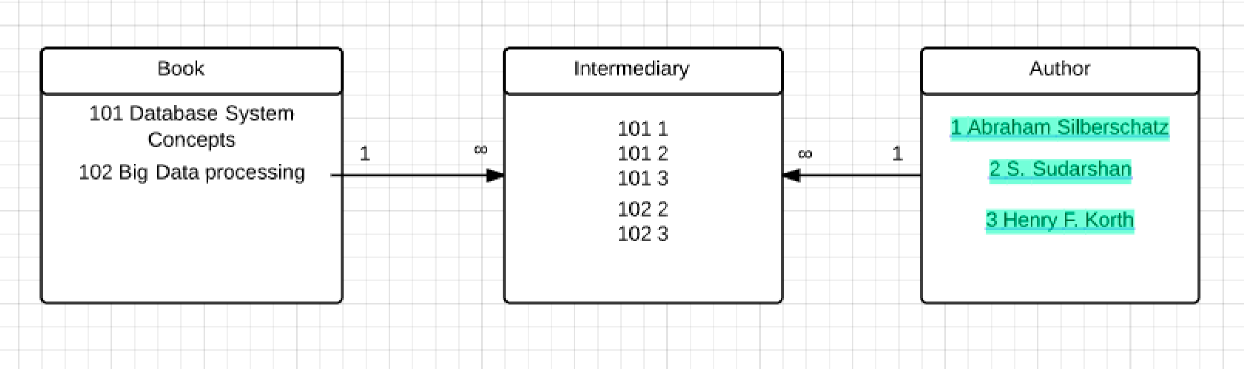


Fig. 10f - many to many relationship

Data Normalization

Data normalization is the process of removing unnecessary redundancy from a database. There are specific steps in data normalization. Each step leads the database to a higher normal form. Data that has not been through the data normalization process is not in normal form.

The first normal form (1NF) ensures that each field is an atomic value. Each field cannot be a collection or set of values, it must be a single value. In other words each row of every column must have only one value, there cannot be more than one value separated by comma or any other delimiter. For example, one book can have many authors but all the authors of one book cannot be stored in the same row neither can we make a separate column to add author 1, author 2 etc.

	A	B	C	D	E	F	G
1	StudentID	StudentNam	StudentEmail	BookName	BookGenre	AuthorName	AuthorDetails
2	1	Stephen	dschmidtn@goo.gl	Database System Concepts	Database	Abraham Silberschatz, S. Sudarshan, Henry F. Korth	Yale University
3	2	will	dschmidtn@goo.gl	Database System Concepts	Database	Abraham Silberschatz - S. Sudarshan - Henry F. Korth	Yale University
4							

Fig. 10g - Data not following 1NF

To make the above data follow 1NF we make separate tables for students, books and authors as shown above and then there is no need to have more than a single value in any row.

For the data to be in **Second normal form (2NF)** it should meet an additional criterion after following 1NF i.e. any non-key attribute should depend entirely on the primary key. 2NF mainly comes into picture when we consider composite primary key as we need to make sure in such cases that any non-key column should be dependent on the entire primary key not on just one column of the composite primary key.

Consider the following example of events table:

Events

Course	Date	CourseTitle	Room	Capacity	Available	...
SQL101	3/1/2013	SQL Fundamentals	4A	12	4	
DB202	3/1/2013	Database Design	7B	14	7	
SQL101	4/14/2013	SQL Fundamentals	7B	14	10	
SQL101	5/28/2013	SQL Fundamentals	12A	8	8	
CS200	4/15/2012	C Programming	4A	12	11	

composite primary key

Fig. 10h - Data not following 2NF

In the above table, we are using a composite primary key of Course and Date. So according to our definition all the non-key attributes i.e. CourseTitle, Room, Capacity, Available should depend on the entire primary key. However, if we take a closer look, CourseTitle depends only on the field Course. Now the question is how can we change it to follow 2NF. The simplest solution is to create

a Course table that contains all the fields for the Course. In our example the Course table would consist of the Course field and the CourseTitle field.

For the data to be in **Third normal form** it should meet an additional criterion after following 2NF i.e. any non-key attribute should not depend on any other non-key attribute. Again using the same above example let's see if there is any non-key field that depends on any other non-key field? Indeed there is, Field Room and Capacity, Capacity is directly dependent on field room. We solve this problem just like we solved the problem of 2NF, by making a new table for the two dependent fields.

This set of rules makes sure that there is minimal redundancy of data while storing. It is a good practice to follow at least 3NF while storing data. There are other sets of rules like 4NF, 5NF, 6NF, BCNF and DKNF but these normal forms are beyond the scope of this book.

Keeping all the above points in mind, the final data model can be:

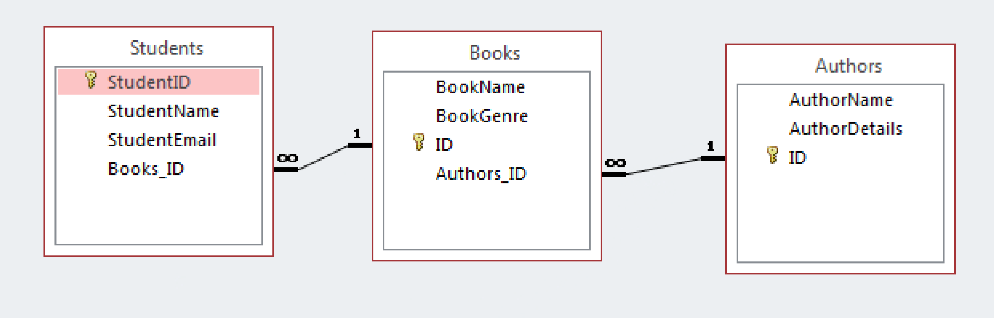


Fig. 10i - Data model 1

Chapter 11

Data insertion through R in SQLite

Relational databases which can be used with R:

SQLite: http://www.sqlite.org/download.html[20] Package for R: RSQlite

MySQL: http://dev.mysql.com/downloads/mysql/ Package for R: RMySQL MySQL workbench: http://dev.mysql.com/downloads/workbench/

PostgreSQL: http://www.postgresql.org/download/ Package for R : RPostgreSQL

Oracle: http://www.oracle.com/technetwork/indexes/downloads/index.html Package for R: ROracle

This chapter limits its examples to the SQLite database, however the other relational database packages are similar to SQLite's functional interface.

Your first step is to download and install the SQLite database. Next install the RSQLite package in R.

```
1  install.packages("RSQLite")
```

After SQLite is installed, you will want to ensure an R session can establish connectivity to a SQLite database. Use the dbConnect() function from the RSQLite package to establish connectivity. The dbConnect accepts two arguments: the first is the type of database, and the second is the name of the database.

```
1  library("RSQLite")
2  db<-dbConnect(SQLite(),dbname="test.sqlite")
3  summary(db)
4
5  <SQLiteConnection>
6    SQLite version:      3.8.6
7    Database name:
8    Loadable extensions:
9    File open flags:
10   VFS:
```

[20] http://www.sqlite.org/download.html

If SQLite is installed correctly you should get a summary like above.

Line 2 above, stores the established connection in the R db variable. The connectivity variable is used every time you access the SQLite server. The types of operations you can perform are: create, read, update and delete. These are known as the CRUD operations.

We will use the example from Chapter 10 and use the same books dataset to create three tables as shown in data model 1 of Fig. 10i. Here is the link[21] of the data again.

Creating tables

We can proceed with the code with two different structures.

First - Create tables(define primary and foreign keys), Manipulate data in R(Segregate data into different variables according to the tables), Insert the data using either INSERT SQL statements (not suitable for big datasets) or Insert data using dbWriteTable function.

Second - Manipulate data in R, Insert data using dbWriteTable which creates a new table according to the value passed in the argument. There is no foreign key constraint in this case thus data manipulation needs to be done with extreme caution. We will see the meaning of foreign key constraint in the coming section.

```
1  > dbSendQuery(conn=db, "CREATE TABLE Authors
2  + (AuthorID INTEGER PRIMARY KEY,
3  + AuthorName TEXT,
4  + AuthorDetails TEXT)")
5  <SQLiteResult>
6
7  > dbSendQuery(conn=db, "CREATE TABLE Books
8  + (BooksID INTEGER PRIMARY KEY,
9  + BooksName TEXT,
10 + BooksGenre TEXT,
11 + AuthorID INTEGER,
12 + FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID))")
13 <SQLiteResult>
14
15 > dbSendQuery(conn=db, "CREATE TABLE Students
16 + (StudentID INTEGER PRIMARY KEY,
17 + StudentName TEXT,
18 + StudentEmail TEXT,
19 + BooksID Integer,
20 + FOREIGN KEY (BooksID) REFERENCES Books(BooksID))")
21 <SQLiteResult>
```

[21] https://drive.google.com/file/d/0B9uiGI8JEJw5X1U5dzNOdEhPcXc/view

Inserting data using INSERT statements

The dbListTables() function returns the list of tables in the database. The dbSendQuery function sends a query to SQLite from R. The first argument is the connection variable, the second argument is the query. The query specifies the operation you wish to perform as well as the criteria for selecting the data for the operation. A simple INSERT query consists of the INSERT keyword followed by the name of the table <tablename> and the values to be stored in the table. The values as well as the data types for the values that are being inserted should match the fields in <tablename>. For example, if the Student table was created with a column name StudentEmail with format TEXT then StudenEmail column can only accommodate TEXT not INTEGERS.

```
> dbListTables(db)
[1] "Authors"  "Books"    "Students"
> dbSendQuery(conn = db, "pragma foreign_keys=on;")
<SQLiteResult>

> dbSendQuery(conn = db,
+   "INSERT INTO Authors
+   VALUES (1, 'Khaled Hosseini', 'Afghan-born American novelist and physician')\
")
<SQLiteResult>

> dbSendQuery(conn = db,
+   "INSERT INTO Authors
+   VALUES (2, 'Abraham Silberschatz', 'Yale University')")
<SQLiteResult>

> dbSendQuery(conn = db,
+   "INSERT INTO Books
+   VALUES (101, 'And the mountains echoed', 'Historical fiction',1)")
<SQLiteResult>

> dbSendQuery(conn = db,
+   "INSERT INTO Students
+   VALUES (1, 'Michelle ', 'meecheli@gmail.com',101)")
<SQLiteResult>
```

In the above code pragma foreign_keys = on is a flag to tell SQLite to turn on the foreign key constraint i.e. you cannot inserta data record with a foreign key that does not exist in the referenced table. This constraint is known as **referential integrity**.

```
> dbSendQuery(conn = db,
+ "INSERT INTO Students
+ VALUES (2, 'Michelle ', 'meecheli@gmail.com',102)")

Error in sqliteSendQuery(conn, statement) :
  rsqlite_query_send: could not execute1: FOREIGN KEY constraint failed
```

The statement above generates an error since there is no BooksId=102 within the database. This means the INSERT statement failed due to the foreign key constraint. Similarly we cannot insert a record without a unique primary key. If the primary key already exists wihin the relation, the INSERT command fails.

```
> dbSendQuery(conn = db,
+ "INSERT INTO Students
+ VALUES (1, 'Michelle ', 'meecheli@gmail.com',101)")

Error in sqliteSendQuery(conn, statement) :
  rsqlite_query_send: could not execute1:
  UNIQUE constraint failed: Students.StudentID
```

Fetching data

We can fetch data using dbReadTable to read the entire table, We can also fetch customized data by querying using the dbFetch function.

```
> dbReadTable(db,"Authors")

  AuthorID          AuthorName                              AuthorDetails
1        1      Khaled Hosseini Afghan-born American novelist and physician
2        2 Abraham Silberschatz                             Yale University

> dbListFields(db,"Authors")
[1] "AuthorID"      "AuthorName"    "AuthorDetails"

#Querying using dbFetch function.
> data<- dbSendQuery(conn = db, "SELECT AuthorName from Authors;")
> dbFetch(data)
            AuthorName
1      Khaled Hosseini
2 Abraham Silberschatz
```

Inserting an entire CSV file

```
1  > database<-dbConnect(SQLite(),dbname="newTest.sqlite")
2  > dbWriteTable(conn = database, name = "BooksData", value = "bookData.csv",
3  + row.names = FALSE, header = TRUE)
4  [1] TRUE
5
6  > dbListFields(database,"BooksData")
7  [1] "StudentID"     "StudentName"   "StudentEmail"  "BookName"
8     "BookGenre"     "AuthorName"    "AuthorDetails"
```

In the above code, we inserted raw data, with a lot of redundancy i.e. data is not in 3NF. We can manipulate the data in R to create different CSV or data frames following the data model of Fig. 10i.

```
1  > db<-dbConnect(SQLite(),dbname="booksData.sqlite")
2  > df<-read.csv("bookData.csv", header=TRUE)
3
4  # Use the unique function to remove redundancy
5  > authors<-unique(cbind.data.frame(as.character(df$AuthorName),
6                               as.character(df$AuthorDetails)))
7  > colnames(authors)<- c("Author Name", "Author Details")
8
9  #Create a new row of Primary keys i.e. AuthorID column
10 > authors$AuthorID<-sample(1:20,nrow(authors))
11
12 # Authors table is ready for insert
13 > dbWriteTable(conn = db, name = "Authors", value = authors,
14 + row.names = FALSE) #Writing Authors table
15 [1] TRUE
16
17 # merge statement to merge the newly created AuthorID with the original dataset
18 > merged.data<- merge(df,authors,by.x=c("AuthorName","AuthorDetails"),
19 + by.y=c("Author Name","Author Details"))
20 # Take a look at the merged data. Above line just added AuthorID fields
21 # to the original data
22
23 # Use the unique function to remove redundancy
24 > books<- unique(cbind.data.frame(as.character(df$BookName),
25                                   as.character(df$BookGenre)))
26 > colnames(books)<- c("Book Name", "Book Genre")
27
28 #Create a new row of Primary keys i.e. booksID column
29 > books$bookID<-sample(100:130,nrow(books))
30
```

```
31  # Again merging the created unique ID to the original dataset
32  # to make sure no data is lost
33  > finalmerge<- merge(merged.data,books, by.x=c("BookName","BookGenre"),
34                                     by.y=c("Book Name","Book Genre"))
35  # Again take a look at the finalmerge vector now it has both
36  # bookID and AuthorID
37
38  # capturing Books data from merged dataframe
39  # above with both authorID and booksID
40  > booksData<-unique(cbind.data.frame(finalmerge$bookID,finalmerge$BookName,
41                                 finalmerge$BookGenre,finalmerge$AuthorID))
42  > colnames(booksData)<- c("BookID","Book Name", "Book Genre", "AuthorID")
43  # We did this step to extract uinque rows of book with AuthorID
44  # so both primary key and foreign key is intact
45
46  # Books table is now ready for insert
47  > dbWriteTable(conn = db, name = "Books", value = booksData,
48  + row.names = FALSE) #Writing Books Table
49  [1] TRUE
50
51  # As student table is the last table to insert we can just generate
52  # unique rows for this table with primary and foreign keys
53  > studentsData<-unique(cbind.data.frame(finalmerge$StudentID,
54      finalmerge$StudentName,finalmerge$StudentEmail,finalmerge$bookID))
55
56  > colnames(studentsData)<- c("StudentID", "Student Name","Student Email"
57                                     , "BookID")
58
59  # Student table is now ready for insert
60  > dbWriteTable(conn = db, name = "Students", value = studentsData,
61  + row.names = FALSE) #Writing Students table
62  [1] TRUE
```

```
# AuthorID column looks a bit messy due to the space constraint.
> head(dbReadTable(db,"Authors"))
      Author.Name                                              Author.Details A\
uthorID
1 Khaled Hosseini                  Afghan-born American novelist and physician  \
     10
2  David Mitchell                                             English novelist  \
     19
3  William gibson  American-Canadian speculative fiction novelist and essayist  \
      1
4    Markus Zusak                                            _Australian writer  \
     14
5     Ken follett              Welsh author of thrillers and historical novels  \
      4
6     Leo Tolstoy Russian novelist regarded as one of the greatest of all time  \
      3

#double check the merge by comparing the data
#I can just see "A thousand splendid suns" and "And the mountains echoed"
#belongs to author Khaled Hosseini with author ID 10.
> head(dbReadTable(db,"Books"))
  BookID                           Book.Name         Book.Genre AuthorID
1    103 A Short History of Nearly Everything        Non-fiction       12
2    111             A thousand splendid suns              Novel       10
3    128                    A walk in the woods        Non-fiction       12
4    108        All quiet on the western front          War novel        2
5    106            And the mountains echoed Historical fiction       10
6    100                        Best war ever        Non-fiction       15

> head(dbReadTable(db,"Students"))
  StudentID Student.Name         Student.Email BookID
1        19        Debra ccastillou@ustream.tv    103
2         4         Emma       emmij@gmail.com    111
3        38         Rose        kgordon17@g.co    128
4        26       Albert       mpalmerv@epa.gov    128
5        15        Albert   mharrism@cpanel.net    108
6        36      Phillip   dpalmer15@sogou.com    108
```

Querying data

Joins can be used to fetch data fields from multiple tables. For example, if we want to recreate the .csv file from the tables, we would JOIN data from the Students, Books nd Authors tables. When we create a JOIN statement we specify the fields that represent the relationships between the entities in

the ON clause of the JOIN statement. In our database, the foreign BookID field stored in the Students table is restricted to be equal to the BookID in the Books table. The same is true for AuthorID, we limit the foreign key found in the Books table to the corresponding AuthorID in the Authors table.

```
1  > query1<- dbSendQuery(db, "Select [Student Name], [Book Name],
2  +  [Author Name] from Students
3  + INNER JOIN Books ON Students.BookID = Books.BookID
4  + INNER JOIN Authors ON Books.AuthorID = Authors.AuthorID")
5  > dbFetch(query1,5)
6    Student Name                            Book Name          Author Name
7  1        Debra  A Short History of Nearly Everything          Bill Bryson
8  2         Emma               A thousand splendid suns     Khaled Hosseini
9  3         Rose                    A walk in the woods          Bill Bryson
10 4       Albert                    A walk in the woods          Bill Bryson
11 5       Albert        All quiet on the western front  Erich Maria Remarque
```

NOTE- When there are spaces within the names of tables or fields, the names should be enclosed within square brackets.

A WHERE clause can be used to create a filter on the returning records. It allows the programmer to specify the specific records from the database it would like to manipulate within the R session. It is always preferable to limit the data sent between the R session (the client) and the database server since it will minimize the amount of data transferred between the two systems; thus leading to a shorter transfer time.

```
1  > query2<- dbSendQuery(db, "Select [Student Name], [Student Email],
2  + [Book Genre], [Author Name] from Students
3  + INNER JOIN Books ON Students.BookID = Books.BookID
4  + INNER JOIN Authors ON Books.AuthorID = Authors.AuthorID
5  + WHERE [Student Name]='Michelle'")
6  > dbFetch(query2)
7    Student Name              Student Email          Book Genre     Author Name
8  1     Michelle          meecheli@gmail.com Historical fiction Khaled Hosseini
9  2     Michelle rcollins1j@wunderground.com         Non-fiction     Naomi Klein
```

The COUNT function counts the number of instances for a field or table.

```
> query3<- dbSendQuery(db, "Select [Author Name], COUNT(*) from Students
+ INNER JOIN Books ON Students.BookID = Books.BookID
+ INNER JOIN Authors ON Books.AuthorID = Authors.AuthorID
+ WHERE [Author Name]='Bill Bryson'")
> dbFetch(query3)
  Author Name COUNT(*)
1 Bill Bryson        6
```

Exercise- Let's try to comprehend the trade-off between space and time with relational databases. In this chapter, we have dealt with a small database. Can you increase this dataset say 15 times and then use the system.time() function as we did in chapter 5 or create a result set from the dataset. Then compare the creation of the same result set from a relational database following 3NF.

The main point of the above exercise is to comprehend the basic trade-off in computer science. We deal with this trade-off on a daily basis where we have to compromise either on the storage space or the time to fetch/retrieve data. With the normal data set, we can retrieve any data of interest very quickly using some sorting operation but it takes a lot of storage space. On the other hand storage of 3NF relational database would require less memory space but executing a complex join could take a long time. So we trade-off between the storage space and the time needed to fetch the data according to the project requirements.

Chapter 12

Retrieving relational data

We have a working relational database and we saw a glimpse of fetching the data from relational databases. In this chapter, we will explore the data retrieval in depth using basic SQL statements through R.

Select statements

Select statements are used to fetch the data and the syntax for select statement is just like reading an English sentence without proper grammar i.e. "SELECT <ColumnName> from <TableName>". Different columns can be retrieved at once using comma as a separator.

```
db<-dbConnect(SQLite(),dbname="booksData.sqlite")
dbGetQuery(db, " SELECT [Student Name], [Student Email] from Students")
```

Line 2 returns the entire Student table. The returned result set can now be processed by an R program. Can we mix the SQL code with R code to limit the number of entries displayed? Yes, we can and that is the advantage of using R. R gives us the freedom to manipulate data after retrieval from the database.

```
> head(dbGetQuery(db, " SELECT [Student Name],
          [Student Email] from Students"),10)
   Student Name          Student Email
1         Debra  ccastillou@ustream.tv
2          Emma        emmij@gmail.com
3         Rose          kgordon17@g.co
4       Albert        mpalmerv@epa.gov
5        Albert    mharrism@cpanel.net
6       Phillip     dpalmer15@sogou.com
7      Michelle      meecheli@gmail.com
8          Adam  jwashingtonr@bbc.co.uk
9         Scott rmedinaf@clickbank.net
10  Stephanie      wfloress@alexa.com
```

Although we can limit the output in SQL as well.

```
1  > dbGetQuery(db, " SELECT StudentID, [Student Name]
2                          from Students LIMIT 10")
3     StudentID Student Name
4  1         19        Debra
5  2          4         Emma
6  3         38        Rose
7  4         26      Albert
8  5         15        Albert
9  6         36     Phillip
10 7          1     Michelle
11 8         22        Adam
12 9          8        Scott
13 10        23   Stephanie
```

> *NOTE*- SQL is not case-sensitive, but it's a good practice to capitalize the keywords of SQL, this helps distinguish the column names and table names from keywords.

Aliases

Typically, it is not best practice to use spaces within column names. However as specified before it can be done by surrounding the column name by square brackets [].

Aliases can be provided for any variable within a result set. It is a mechanism for customizing the result set.

```
1  > dbGetQuery(db, " SELECT StudentID AS [Student ID],
2      [Student Name] AS StudentName from Students LIMIT 10")
3
4     Student ID StudentName
5  1          19       Debra
6  2           4        Emma
7  3          38       Rose
8  4          26     Albert
9  5          15       Albert
10 6          36    Phillip
11 7           1    Michelle
12 8          22       Adam
13 9           8       Scott
14 10         23   Stephanie
```

WHERE Clause

The WHERE clause can be used to filter the queried result with a specific condition. The syntax in this case is just an extension of select statement i.e. " SELECT <columnName> from <tableName> WHERE <condition>".

```
1  > dbGetQuery(db, " SELECT StudentID,[Student Name],
2      [Student Email] from Students WHERE [Student Name]='Michelle'")
3
4    StudentID Student Name              Student Email
5  1         1     Michelle          meecheli@gmail.com
6  2        45     Michelle rcollins1j@wunderground.com
```

GROUP BY clause

The GROUP BY clause is used to group records that have the same value together. Each value found within the column creates a separate group and each aggregated function is performed on each group.

Syntax:" SELECT <columnName>,aggregate_functions(<columnName>) from <tableName> GROUP BY <columnName>"

```
1  > dbGetQuery(db, "SELECT [Book Genre],
2    Count(*) AS [Number of Books] from Books GROUP BY [Book Genre] ")
3
4              Book Genre Number of Books
5  1            Adventure               1
6  2             Database               1
7  3                Drama               1
8  4              Fiction               2
9  5    Historical fiction               3
10 6              History               1
11 7               Humour               1
12 8          Non-fiction               9
13 9                Novel               1
14 10       Romance Novel               1
15 11     Science fiction               3
16 12   Travel literature               1
17 13           War novel               1
18 14 Young-adult fiction               1
19 15          psychology               1
```

Aggregate functions like SUM, COUNT, MAX, MIN, FIRST, LAST can be used on any column and using the alias with the same gives a better presentation.

ORDER BY clause

The ORDER BY clause is used to arrange the queried result in ascending or descending order based on a certain column. The ORDER BY function is just a method to sort data. The syntax includes specifying the selected columns and then specifying the "order by" on which column and ascending or descending. ASC and DESC are the keywords for ascending and descending.

```
1  > dbGetQuery(db, "SELECT [Book Genre],
2      Count(*) AS [Number of Books] from Books
3      GROUP BY [Book Genre]
4      ORDER BY [Book Genre] DESC")
5
6             Book Genre Number of Books
7  1          psychology               1
8  2  Young-adult fiction              1
9  3            War novel              1
10 4    Travel literature              1
11 5      Science fiction              3
12 6        Romance Novel              1
13 7                Novel              1
14 8          Non-fiction              9
15 9               Humour              1
16 10             History              1
17 11  Historical fiction              3
18 12             Fiction              2
19 13               Drama              1
20 14            Database              1
21 15           Adventure              1
```

HAVING clause

The HAVING clause specifies a filtering criteria for the GROUP BY results. It provides the same functionality as the WHERE clause except on the aggregated results. The syntax for the HAVING clause is similar to the syntax for the WHERE with the exception that the HAVING clause can specify aggregated functions in the conditional clause.

```
1  > dbGetQuery(db, "SELECT [Book Genre],
2      Count(*) AS [Number of Books] from Books
3          GROUP BY [Book Genre]
4          HAVING Count(*) >5")
5
6    Book Genre Number of Books
7  1 Non-fiction               9
```

JOINS

Joins are the most commonly used function when it comes to retrieving data from a relational database. A JOIN statement connects data from different tables and fetch the data across tables. There are four types of Joins:

1. INNER JOIN
2. LEFT OUTER JOIN
3. RIGHT OUTER JOIN
4. FULL OUTER JOIN OR CROSS JOIN

INNER JOIN

Inner join returns only the instances which occur at least once in both the tables. A specific type of an INNER JOIN is called a NATURAL JOIN. A NATURAL JOIN completes an INNER JOIN between two tables on the fields that have the same name.

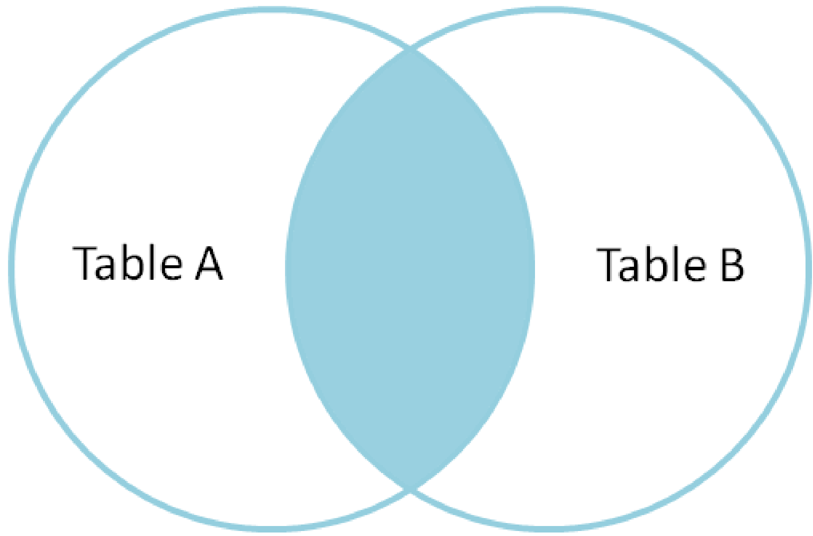


Fig. 12a - Inner Join representation

```
1  > dbGetQuery(db, "Select [Book Name],[Author Name],
2                  COUNT(*) AS [Issued By] from Students
3  + INNER JOIN Books ON Students.BookID = Books.BookID
4  + INNER JOIN Authors ON Books.AuthorID = Authors.AuthorID
5  + GROUP BY [Book Name]")
6
7                              Book Name          Author Name Issued By
8  1  A Short History of Nearly Everything          Bill Bryson         1
9  2             A thousand splendid suns       Khaled Hosseini        1
10 3                    A walk in the woods          Bill Bryson         2
11 4          All quiet on the western front Erich Maria Remarque         2
12 5             And the mountains echoed       Khaled Hosseini         3
13 6                        Best war ever          John Stauber         3
14 7                                Blink      Malcolm Gladwell         2
15 8                          Cloud atlas        David Mitchell         2
```

```
16  9     Database System Concepts Abraham Silberschatz       1
17  10                 Down under             Bill Bryson     1
18  11               Going clear        Lawrence Wright       2
19  12               Kite runner        Khaled Hosseini       2
20  13               Neuromancer         William gibson       2
21  14                   No logo            Naomi Klein       1
22  15                Outlander          Diana Gabaldon       3
23  16                 Outliers        Malcolm Gladwell       2
24  17             Shadow Divers          Robert Kurson       1
25  18           The bone clocks         David Mitchell       1
26  19            The book thief           Markus Zusak       1
27  20      The elegant universe           Brian Greene       2
28  21        The lost continent            Bill Bryson       2
29  22            The peripheral         William gibson       3
30  23       The pillars of earth            Ken follett      1
31  24                 The prize          Daniel Yergin       2
32  25    This changes everything           Naomi Klein       1
33  26                   Voyager         Diana Gabaldon       2
34  27             War and Peace            Leo Tolstoy       2
35  28            pirate hunters          Robert Kurson       2
```

LEFT OUTER JOIN

Left join returns all the instance from the left table and the matched rows from the right table.

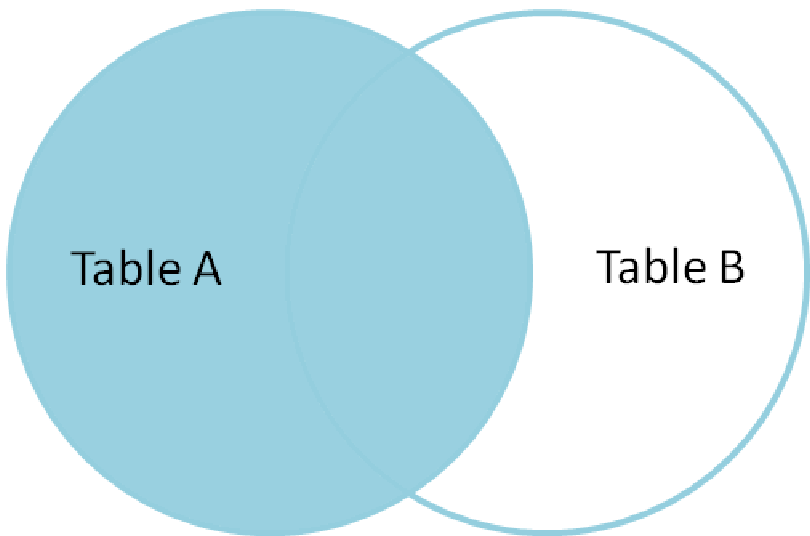


Fig. 12b - LEFT OUTER JOIN representation

```
1  > db1<-dbConnect(SQLite(),dbname="orders.sqlite")
2
3  > dbSendQuery(conn=db1, "CREATE TABLE customers
4  + (customerID INTEGER PRIMARY KEY,
5  + customerName TEXT,
6  + Address TEXT)")
7  <SQLiteResult>
8
9  > dbSendQuery(conn=db1, "CREATE TABLE orders
10 + (orderID INTEGER PRIMARY KEY,
11 + customerID INTEGER,
12 + orderDetails TEXT,
13 + price TEXT,
14 + FOREIGN KEY (customerID) REFERENCES customers(customerID))")
15 <SQLiteResult>
16
17 > dbSendQuery(conn = db1,
18 + "INSERT INTO customers
19 + VALUES (1, 'Latika', 'Boston')")
20 <SQLiteResult>
21
22 > dbSendQuery(conn = db1,
23 + "INSERT INTO customers
24 + VALUES (2, 'John', 'California')")
25 <SQLiteResult>
26
27 > dbSendQuery(conn = db1,
28 + "INSERT INTO customers
29 + VALUES (3, 'Elton', 'New York')")
30 <SQLiteResult>
31
32 > dbSendQuery(conn = db1,
33 + "INSERT INTO orders
34 + VALUES (101,1, 'Nexus 6', '$550')")
35 <SQLiteResult>
36
37 > dbSendQuery(conn = db1,
38 + "INSERT INTO orders
39 + VALUES (102,1, 'Nexus 6 smart cover', '$15')")
40 <SQLiteResult>
41
42 > dbGetQuery(db1, "Select customerName, orderDetails, price from customers
```

```
43 + LEFT JOIN orders ON customers.customerID = orders.customerID")
44
45   customerName        orderDetails price
46 1       Latika             Nexus 6   $550
47 2       Latika Nexus 6 smart cover    $15
48 3         John                <NA>   <NA>
49 4        Elton                <NA>   <NA>
50
51 > dbGetQuery(db1, "Select customerName, orderDetails, price from customers
52 + INNER JOIN orders ON customers.customerID = orders.customerID")
53   customerName        orderDetails price
54 1       Latika             Nexus 6   $550
55 2       Latika Nexus 6 smart cover    $15
```

> **Note** - Both Right joins and outer joins are not implemented within many databases including SQLite. The example codes for these two will not be executable.

RIGHT OUTER JOIN

Just the opposite of the LEFT OUTER JOIN, the RIGHT OUTER JOIN returns all the instances of the right table as well as the matching rows from the left table.

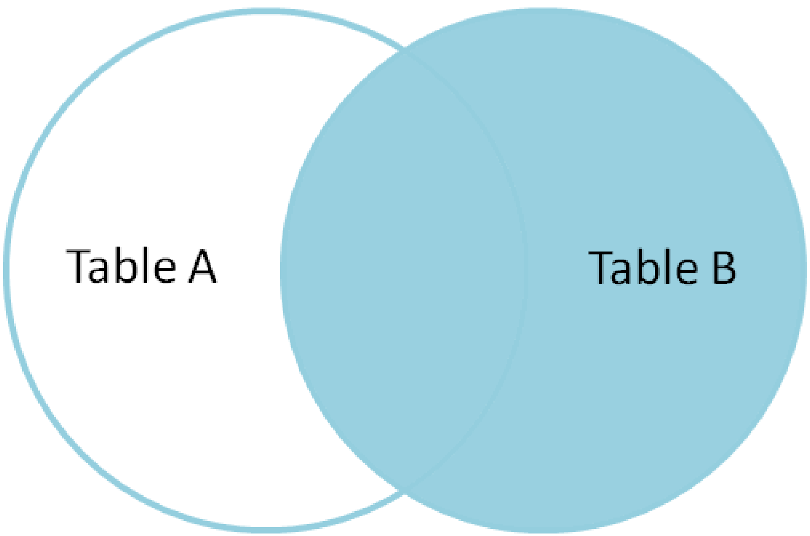


Fig. 12c - RIGHT OUTER JOIN representation

```
1 dbGetQuery(db1, "Select orderDetails, price, customerName, Address from orders
2            RIGHT JOIN customers ON orders.customerID = customers.customerID")
```

CROSS JOIN OR THE FULL OUTER JOIN

The FULL OUTER JOIN returns all the instances and the variables from both tables; if Table A has M records and table B has N records then the result set contains M*N records. It is analagous with the CROSS PRODUCT of the two tables.

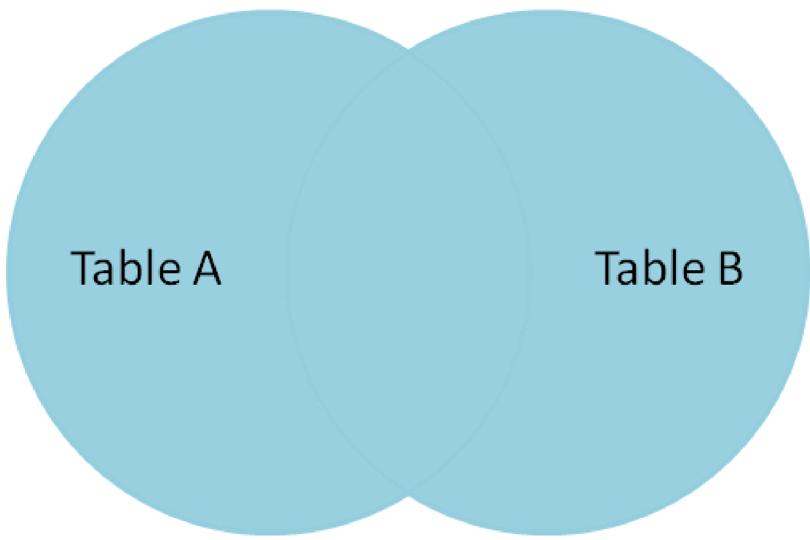


Fig. 12d - CROSS JOIN OR THE FULL OUTER JOIN representation

```
dbSendQuery(conn = db1,
            "INSERT INTO orders
            VALUES (103,4, 'iphone cover', '$25')")
dbSendQuery(conn = db1,
            "INSERT INTO orders
            VALUES (104,5, 'samsung galaxy edge', '$750')")
dbGetQuery(db1, "Select customerName, orderDetails, price from customers
           OUTER JOIN orders ON customers.customerID = orders.customerID")
```

Chapter 13

Non-relational databases

Introduction

Relational Database Management System (RDBMS) is by far the most common data storage mechanism. In fact prior to 2004 almost all the data was stored in warehouses based on RDBMS. The real downside of RDBMS came into the picture when databases grew to be in the petabytes.

The following is a list of common RDMS problems:

1. SQL is by far the most common method of fetching data from RDBMS. When the database size reaches sizes in the terabyte and petabyte range, then the SQL queries can take a very long time to execute.
2. Before storing the data, we need to have a complete picture about the structure and organization of data so that it is easy to retrieve data later on.
3. Infrastructure requirements to handle petabytes of data with RDBMS become enormous with the need for specialized servers to prevent loss of information.

The above challenges were too much of an investment with little return in functionality and smooth operational processes. This lead to an evaluation of the functionality provided by a RDMS and the needed functionality for storing and retrieving large databases in an efficient manner. This led to the No SQL revolution or the Non-relational databases. Google was the first one to come up with their database called “Big Table” in 2004. The research paper of this project became the foundation of Non-relational databases and in 2008 Yahoo announced the implementation of Hadoop.

Non-relational databases are also known as NoSQL databases, the NoSQL stands for NOT ONLY SQL. It is referring to the myriad of data models supported by these databases. In a relational database you are limited to representing your data objects as a two dimensional table. The Not Only SQL databases allow you to use other data models such as: key value pair, document style model, network based model, graph model, hierarchical model etc.

Other common features of NoSQL databases:

1. **Cost effective**: NoSQL databases don’t require specialized servers to maintain data, in fact, usage is as inexepensive as 10% - 20% of the specialized RDBMS servers.

2. **No structure needed**: One of the most important features of NoSQL databases is the ability to store unstructured data. Based on the flavor of NoSQL database used one can store almost anything together. Any type of data like weblogs, social networking text files, satellite images, graphs, biological data like 3d analysis of molecules or any other kind of data can be stored in the same data file.
3. **Massively parallel processing**: NoSQL databases use the concept of massively parallel processing to improve the execution time of data access. The NoSQL databases use many off the shelf simple processors working in parallel to accomplish a specific data access. The data and operations are spread across multiple nodes; and the work to complete the data access operation is done in parallel.
4. **Object-oriented programming**: Object-oriented programming makes it simple and flexible to code with NoSQL databases.
5. **Tolerance to disk failure**: NoSQL databases were designed to handle petabytes of data. There are specific algorithms that can keep the database system available even when a disk failure occurs. The data is divided into chunks and are duplicated on other disks. The system manages the duplicated copies of the data; and will use a different replicated copy of the data from a different node, if it is more optimal to do so or if that node is not available..
6. **Simpler database management**: A database manager for a SQL database is a highly trained engineer who understands the optimizations provided by a RDMS and can utilize these optimizations for a database. NoSQL databases have automated many of the tasks associated with a database administer (DBA). The goal is to simplify or potentially eliminate a DBA for NoSQL databases.

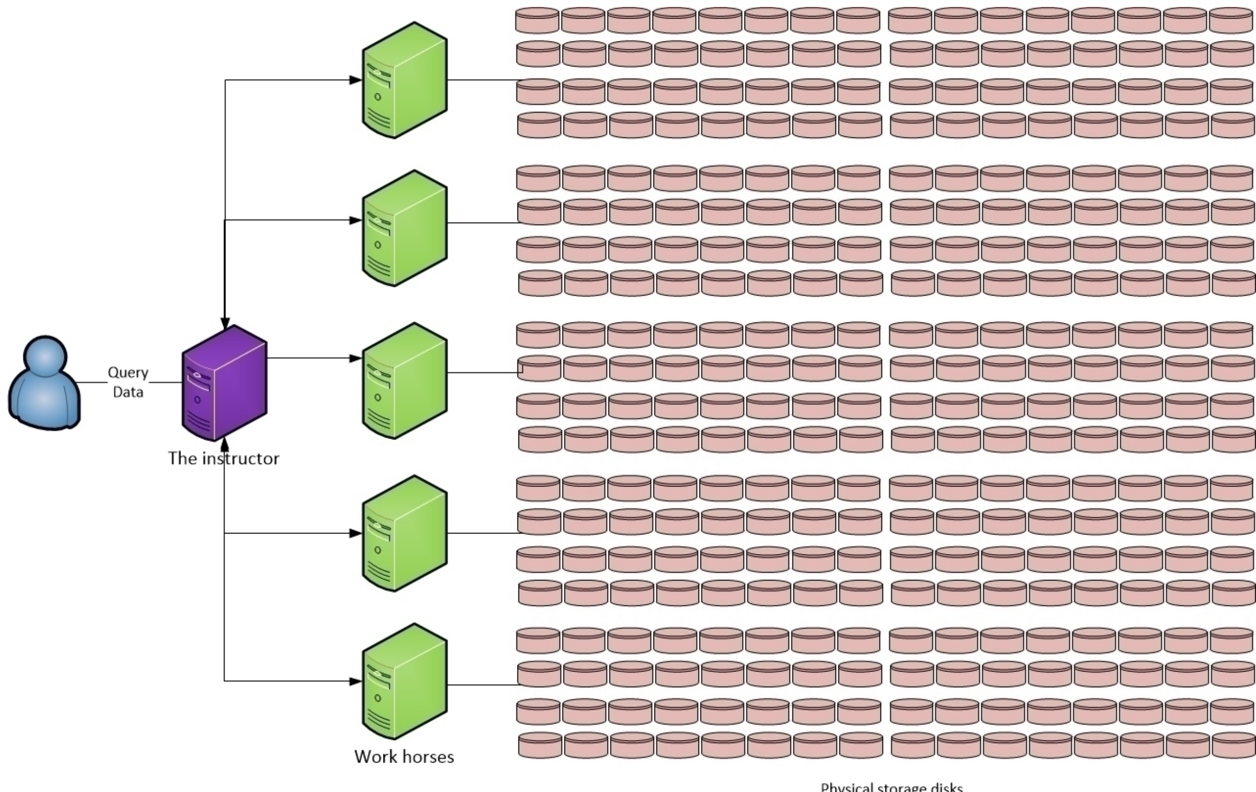


Fig. 13a - Massive parallel processing

Types of NoSQL databases

NoSQL databases support many different data models. Below is a list of some models supported by NoSQL.

1. Key-value databases eg riak, project Voldemort
2. Document oriented databases eg MongoDB, CouchDB
3. Columnar databases eg Cassandra
4. Graph database eg Neo4J

Key-value databases

Key-value paired databases are the simplest model supported by the NoSQL databases. They associate a data value with a specific key value. The key value must be known for retrieval of the data value. Key-value pair databases are similar to a hash table where access to a hash entry is via the hash key. The key can be synthetic or auto-generated and the values associated with the key can be a simple integer, a string, JSON object, or a BLOB (binary large object).

When an element is being inserted into a key-value pair database, the only restriction on the element is that the key does not exist within the collection of entities found in the database. There is no

restriction of the value of the element, such as size or type restriction. Now the question is what is the meaning of collection of entities here? To understand this let us assume that we have a database of customers placing different orders for different products say data from Amazon. Now we know the schema for this database if we store the data in a relational database, this will require to normalize the data to reduce redundancy. However, when we store the same data in a key-value database, we make a collection of each real world entity.

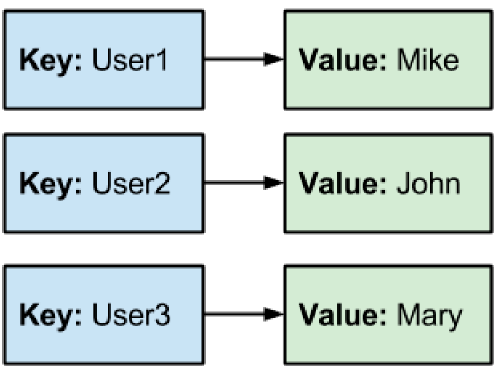


Fig. 13b - Key-value databases

Document oriented databases

Document oriented databases also have the concept of a key to access a particular object. However, the data portion of the object can have a structure. The structure of the object is embedded within the object itself. There is no separate data dictionary as found within a SQL database. In order to understand the structure of an object you retrieve and read the object. The representation of the data on disk typically will use a language such as XML, JSON or BSON (binary JSON) since these languages provide the self-referencing aspect needed by these databases. These languages allow the actual structure of each object stored in a collection of objects to have a different collection of variables stored within it. This is known as a schema-less database. This provides great flexibility when storing data and designing a database. It also allows the database to evolve over time.

When we are retrieving data, we provide an implicit schema i.e. fetching a particular price of a particular order we are assuming that order has a price field.

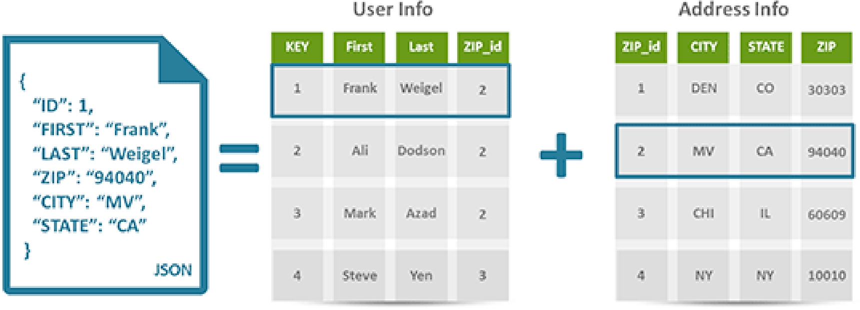


Fig. 13c - document databases

We will see the usage of MongoDB a document oriented database in detail and explore how to store, manipulate and fetch data in MongoDB.

Columnar databases

As the name suggests, a columnar database stores data in columns as oppose to rows. This storage mechanism is useful when most queries access a small subset of the columns in a row. Columns are grouped into families. Typically a column family corresponds to a real world object.

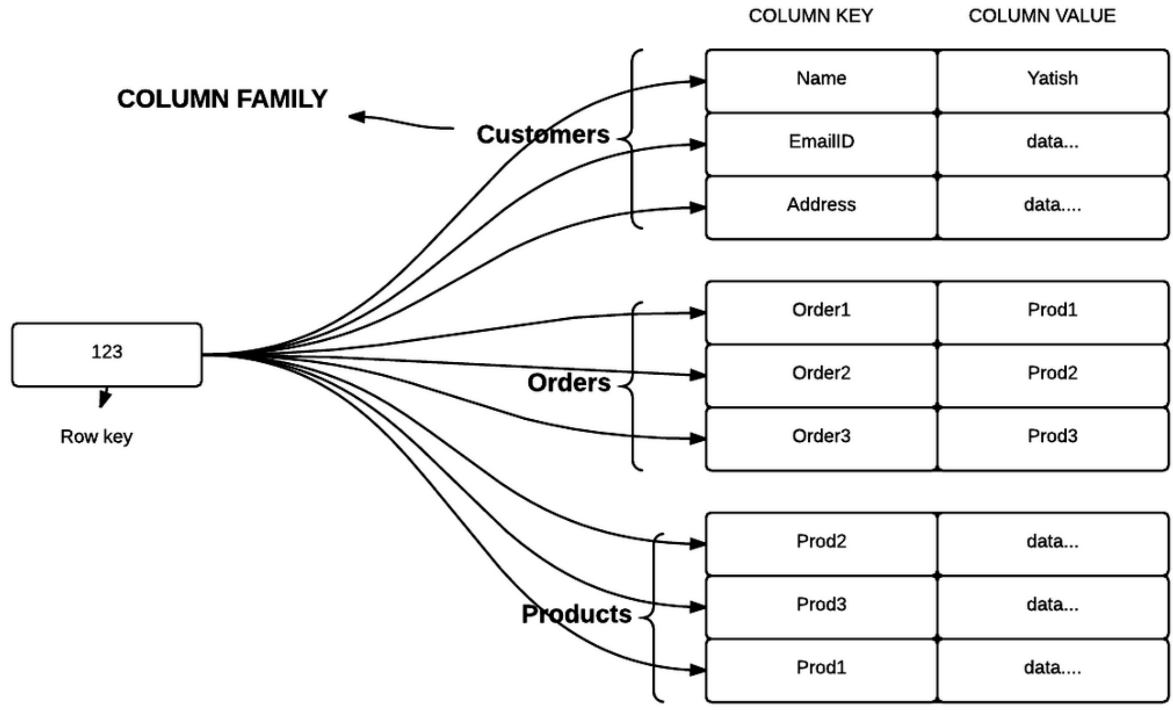


Fig. 13d - Columnar databases

The Figure above shows a representation of a columnar database. This can be compared to a row-oriented database configuration where the fields for an object are stored in the same file. Above data would be stored in the following format in row oriented databases:

{123:{{Name:Yatish},{EmailID:data},{Address:data}},{},{}....}

The same data is stored in columnar databases as:

123:{{Name,EmailID,Address:Yatish,data,data},{},{}....}

Compression is heavily applied in a columnar database since the data stored in each file contains the values for only one field. These fields can be stored in canonical order (such as alphabetical order for strings) and only the difference between the values need to be stored. This leads to fewer bytes per entry. Compression of data can be done using a concept called tokenization. A column may have many repeated values, we replace these repeated values with tokens, this concept of tokenization works better in columnar database as compared to row-oriented database as we do this compression block by block and in columnar database all column values which are bound to be similar are grouped together and hence stored in one block.

Graph Database

Graphs consists of nodes and edges. Typically the node represent entities and the edges represent a relationship between the nodes. A graph database can represent highly interconnected entities fairly easily. Examples of highly interconnected data are data from a social networking site, web references on the web, word usage within articles etc. Graph databases can also be used for applying inference to the network.

Chapter 14

MongoDB

MongoDB is a NoSQL document oriented database. It can be used to store large semi-structured data as an object. A database consists of 0 to many collections. A collection represents a collection of similar real world projects. An object or a document consists of named fields and values. The values of the fields may in turn consist of objects or a collection of objects.

MongoDB can be downloaded from the official site directly. MongoDB[22]

Installation on mac

Step 1: Download the compressed file from the link above or use the terminal to download the file to your current working directory using the following command:

```
curl -O https://fastdl.mongodb.org/osx/mongodb-osx-x86_64-3.0.5.tgz
```

Step 2: Unzip the file by either double clicking on the zipped file or using the following command in terminal:

```
tar -zxvf mongodb-osx-x86_64-3.0.5.tgz
```

Step 3: Copy the extracted files to a new parent folder named mongodb to make it a target directory using the following commands in terminal:

```
mkdir -p mongodb
cp -R -n mongodb-osx-x86_64-3.0.5/ mongodb
```

> Note - Make sure you specify the path of the downloaded file if it is not in your current working directory.

Step 4: Last step is to ensure that the location of mongodb path is exported in the PATH environment variable so that we can directly run any command directly in the terminal without actually going to the corresponding path.

[22] https://www.mongodb.org/downloads

```
export PATH=<mongodb-install-directory>/bin:$PATH
```

To find out the exact path of the current working directory we can use pwd command. My mongodb-install-directory looks like this:

```
export PATH=/Users/Yatish/mongodb/bin:$PATH
```

> Note: Be careful that there should be no space in the command line after typing PATH=

The above command will export this mongodb path to PATH variable only for this session and once you close the terminal and open it again you will have to run this command again to export the path, a better way to avoid this process is to create a .bashrc file or edit the .bashrc file if you already have one and export the PATH there and source the .bashrc file while opening terminal.

Here is how my .bashrc file looks like, where I have exported the path of MySQL, Perl and MongoDB installed and different places on my system.

```
export DYLD_LIBRARY_PATH=/usr/local/bin/mysql/lib

export PATH=/usr/local/ActivePerl-5.20/bin:$PATH
export PATH=/usr/local/ActivePerl-5.20/site/lib:$PATH
export PATH=/Users/Yatish/mongodb/bin:$PATH
export PATH
```

Fig. 14a - .bashrc file

Installation on windows

Step 1: Determine which mongoDb build is required for your system by using the following command in command prompt:

```
wmic os get caption
wmic os get osarchitecture
```

Download the mongodb 32 bit of 64 bit version accordingly.

Step 2: For an interactive installation, locate the .msi file in your Downloads folder and double click on it to run it. It you do not wish to use the .msi file use the following commands in a command prompt shell.

```
msiexec.exe /q /i mongodb-win32-x86_64-2008plus-ssl-3.0.5-signed.msi ^
            INSTALLLOCATION="C:\mongodb" ^
            ADDLOCAL="all"
```

> Note - Before running the above command make sure you are in the directory where the above .msi file is located.

Running MongoDB on windows

Step 1: Make directory to setup mongodb environment:

```
md \data\db
md \data\log
```

Step 2: Start MongoDB server by running the following command:

```
C:\mongodb\bin\mongod.exe
```

> Note- the above command assumes that you have downloaded and installed mongodb in C:/mongodb

Step 3: Run a client that connects to the MongoDB server. This will start the MongoDB shell.

```
C:\mongodb\bin\mongo.exe
```

Step 4: Start using mongodb

Running MongoDB on mac

Step 1: Create a data directory using the following command:

```
mkdir -p /data/db
mkdir -p /data/log
```

Step 2: Set correct permissions for the above-created data directory using the following command:

```
sudo chown `id -u` /data/db
```

Step 3: Run mongodb server by using the following command in terminal:

```
1 mongod
```

Once mongod, the MongoDB server, is started, it is ready to accept client connections. You can terminate the Mongod server by pressing CTRL+C simultaneously in the window running the server.

For our first MongoDB exercise, we will insert the geographic coordinates for U.S. farmers markets for 2013 and write queries to retrieve subsets of the data. The farmer's markets data can be accessed from this link[23].

Before actually going further into the demonstration it is advisable to read the description of the file given in the second row and look at the data in general.

> *NOTE* - Before trying to establish a connection using R make sure that mongodb server is started in the terminal and is ready to accept the connection. To start the server run mongod in the terminal in mac and execute mongod.exe in window as administrator.

Reading XLSX file in R

```
1 library(openxlsx) #load package
2
3 data<-read.xlsx("2013 Geographic Coordinate Spreadsheet for U S  Farmers
4                                 Markets 8'3'1013.xlsx", sheet=1, startRow=3)
5 str(data)
```

In the above command, we set startRow=3 since the market data starts on the third row. We wish to not include the data labels and descriptions found in the first 2 rows.

Connecting to mongodb using R

There are mainly two packages of R used to connect to mongodb namely mongolite and rmongodb. In this book, we demonstrate the functions, objects and object methods in mongolite.

```
1 install.packages("mongolite")
2 library("mongolite")
3
4 mongoData<- mongo("data") #establish connection using mongo command
```

The mongo() function in the mongolite package establishes a connection between the MongoDB "data" collection and the R session. Once you run the above command you should see a message in the terminal window stating connection accepted as shown in the figure below.

[23] https://drive.google.com/file/d/0B9uiGI8JEJw5aWVtVWNzd1BSTk0/view

```
Yatishs-MacBook-Air:~ Yatish$ source ~/.bashrc
Yatishs-MacBook-Air:~ Yatish$ mongod
2015-08-19T10:26:58.539-0400 I JOURNAL  [initandlisten] journal dir=/data/db/journal
2015-08-19T10:26:58.540-0400 I JOURNAL  [initandlisten] recover : no journal files present, no recovery needed
2015-08-19T10:26:58.560-0400 I JOURNAL  [durability] Durability thread started
2015-08-19T10:26:58.560-0400 I CONTROL  [initandlisten] MongoDB starting : pid=7501 port=27017 dbpath=/data/db 64-bit host=Yatishs-MacBook-Air.local
2015-08-19T10:26:58.560-0400 I CONTROL  [initandlisten]
2015-08-19T10:26:58.560-0400 I CONTROL  [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
2015-08-19T10:26:58.560-0400 I JOURNAL  [journal writer] Journal writer thread started
2015-08-19T10:26:58.560-0400 I CONTROL  [initandlisten] db version v3.0.5
2015-08-19T10:26:58.560-0400 I CONTROL  [initandlisten] git version: 8bc4ae20708dbb493cb09338d9e7be6698e4a3a3
2015-08-19T10:26:58.561-0400 I CONTROL  [initandlisten] build info: Darwin bs-osx108-6 12.5.0 Darwin Kernel Version 12.5.0: Sun Sep 29 13:33:47 PDT 2013; root:xnu-20
50.48.12~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2015-08-19T10:26:58.561-0400 I CONTROL  [initandlisten] allocator: system
2015-08-19T10:26:58.561-0400 I CONTROL  [initandlisten] options: {}
2015-08-19T10:26:58.896-0400 I NETWORK  [initandlisten] waiting for connections on port 27017
2015-08-19T10:27:24.971-0400 I NETWORK  [initandlisten] connection accepted from 127.0.0.1:55182 #1 (1 connection now open)
```

Fig. 14b - mongodb connection accepted

Once the connection is established, we can insert data using the insert method of the mongo connection. The method inserts objects into the connected "data" collection.

```
1 > mongoData$insert(data)
2 Complete! Processed total of 8144 rows.
3 [1] TRUE
```

The advantage of using mongolite is that you can directly insert an R dataframe into a Mongo database. This is not true for rmongodb, since with rmongodb the data must be converted to a JSON object before it can be stored to a Mongo database.

The above command will create a default binary file in the mongodb/data directory named test.0 with the data stored as a BSON object. It is essential to know how the data is stored as we will be using the exact same format to fetch the data.

A JSON object is surrounded by curly braces. Each objects consists of field value pairs, where the name of the field an the value of the field are separated by a colon (:).

Functionalities of mongolite

export()

The export() function exports the data to the named external file, as mentioned above, the data is stored in the mongodb/data directory as a binary file. It is good practice to review the newly created file for potential problems.

```
1 # Change the directory to your desktop.
2 # So that you can open this exported file easily.
3
4 > mongoData$export(file("data.txt"))
5 Done! Exported a total of 8144 lines.
```

A JSON file encapsulates each object within curly braces {}. A special field is automatically created for each inserted object. It is the key for the object the _id field; the system ensures it has a unique value. All the column or field names are to the left of the colon and the value is to the right of the colon. Different key value pairs are separated by comma.

Once you understand this structure you can easily use different functionalities of mongolite package.

count()

```
> mongoData$count()
[1] 8144

# count data with specific condition

# count data where value of Vegetables is Y
> mongoData$count('{"Vegetables":"Y"}')
[1] 4326
```

In the above count command, the first argument in curly braces is the column name and the second argument after the colon is the value of that column.

find()

The find method allows you to query the connected data collection. It returns the documents that satisfy the provided criterion.

```
> Cheese <- mongoData$find('{"Cheese":"Y"}')
 Imported 2257 records. Simplifying into dataframe...

> eggsAndVegetables<-mongoData$find('{"Eggs":"Y","Vegetables":"Y"}')
 Imported 3123 records. Simplifying into dataframe...
```

We can refine the find statement by combining many conditions in one statement. The conditions are separated by commas.

sort()

The sort argument to the find method, orders the result set either in ascending or descending order. It is analogous to the ORDER BY clause in SQL. It provides a JSON object that consists of the field to sort by and a data value equal to -1 or 1. A data value of 1 specifies to sort the documents in ascending order; a value of -1 specifies to sort the documents in descending order.

```
Cheese <- mongoData$find('{"Cheese":"Y"}', sort='{"FMID":-1}')
head(Cheese)
```

Fetching specific columns

The find() method also accepts an argument that will limit the columns returned in the result set. Like the sort argument, it accepts a JSON object, where you can specify the fields to be returned in the result set. A data value of 0 means do not return the field, a data value of 1 means the field should be returned in the result set. The default is to return a field unless the 0 value is specified. This example shows the _id field created by MongoDB when a record is inserted into a collection. MongoDB ensures a unique value for each document or object inserted into a collection. In this example, it is displayed as a large hexadecimal number.

```
> dat <- mongoData$find('{"Cheese":"Y"}', fields = '{"FMID":1, "Cheese":1}')
 Imported 2257 records. Simplifying into dataframe...
> head(dat)
                                                  _id    FMID Cheese
1 55, d4, 9b, f4, ce, a0, 6b, 1b, 60, 3a, 06, 92 1005969      Y
2 55, d4, 9b, f4, ce, a0, 6b, 1b, 60, 3a, 06, 93 1008044      Y
3 55, d4, 9b, f4, ce, a0, 6b, 1b, 60, 3a, 06, 94 1000618      Y
4 55, d4, 9b, f4, ce, a0, 6b, 1b, 60, 3a, 06, 97 1008071      Y
5 55, d4, 9b, f4, ce, a0, 6b, 1b, 60, 3a, 06, 9c 1000709      Y
6 55, d4, 9b, f4, ce, a0, 6b, 1b, 60, 3a, 06, 9d 1003233      Y
```

Here is an example where we remove the _id field from the result set.

```
> dat <- mongoData$find('{"Cheese":"Y"}', fields = '{"_id":0,"FMID":1, "Cheese":\
1}')
 Imported 2257 records. Simplifying into dataframe...
> head(dat)
     FMID Cheese
1 1005969      Y
2 1008044      Y
3 1000618      Y
4 1008071      Y
5 1000709      Y
6 1003233      Y
```

distinct()

The distinct method can be used to select the unique values of a column. It is analogous to the DISTINCT keyword in SQL.

```
1 > cities<-mongoData$distinct("city")
2 > head(cities)
3 [1] "Virginia Beach" "Douglasville"   "Kalamazoo"      "New York"       "Wilming\
4 ton"      "Washington"
```

aggregate()

The aggregate method can be used to perform aggregated operations on the qualifying documents. You can specify a field to group the documents by as well as the aggregation function(s). The aggregation functions supported are: count, min, max, sum, average; these are the functions supported by SQL. The aggregate method provides the GROUP BY and aggregation functionality provided by SQL. This example also displays the aliasing functionality found within SQL. In our example we are renaming the State field to _id and the aggregated sum to "count".

```
1 > mongoData$aggregate('[{"$group":{"_id":"$State", "count": {"$sum":1}}}]')
2  Imported 55 records. Simplifying into dataframe...
3                    _id count
4 1            Miinesota     1
5 2              Wyoming    41
6 3           Calafornia     5
7 4                 Utah    40
8 5           California   754
```

More than one aggregate functions can be used together:

```
1 > mongoData$aggregate('[{"$group":{"_id":"$State", "count": {"$sum":1},"max":{"$\
2 max":"$zip"}}}]')
3  Imported 55 records. Simplifying into dataframe...
4                    _id count    max
5 1            Miinesota     1   <NA>
6 2              Wyoming    41  83127
7 3           Calafornia     5  94114
8 4                 Utah    40  84775
9 5           California   754  96150
```

drop()

The drop method removes the connected collection from the current database.

```
> mongoData$drop()
[1] TRUE
```

Limitation of using mongoDB with R

As stated in Chapter 2, there are limitations to the characters that can be within an R variable name. One such limitation is an R variable does not allow a space. In order to easily convert mongoDB field names to R variable names, R converts all spaces to periods. In the example below, the field name "Update Time" is changed to "Update.Time". However, the period is a special character for JSON objects. It specifies that the field is not a simple field but a field that also contains subfields. To avoid this problem make sure you clean your field names by removing periods and spaces BEFORE storing the data to mongoDB.

```
> str(data[45])
'data.frame':       8144 obs. of  1 variable:
 $ Update.Time: chr  "41034.74790509259" "41108.577175925922" "41456.97906250000\
2" "40969.443310185183" ...
```

Here is an example when the programmer has not removed the spaces from the field names. The data cannot be retrieved using the find method, since it assumes that Time is a subfield to the Update field.

```
> mongoData$count('{"Update.Time":"41108.577175925922"}')
[1] 0
> mongoData$find('{"Update.Time":"41108.577175925922"}')
 Imported 0 records. Simplifying into dataframe...
data frame with 0 columns and 0 rows
```

To deal with this limitation a simple solution is to follow camelCasing in the column names of your data.

Stopping the MongoDB server

It is very important to note that once you start a mongoDb server it is using a particular port for the MongoDB server to client communication. If you close the terminal without closing the MongoDb session, then that port is not available to restart a MongoDb session. To stop the server, just press the CTRL key while simultaneously pressing the C key <CTRL+C> in your terminal window. This is not an issue on Windows operating systems since all system resources are deallocated when the command prompt is terminated. However on a Mac system, ensure you stop the MongoDB server with the<CTRL+C> keys or else the server will not be able to restart.

Figure 14.c displays the error when the MongoDB server is not properly terminated:

```
Yatish — bash — 105×28
Last login: Fri Sep 11 15:49:12 on ttys000
Yatishs-MacBook-Air:~ Yatish$ source ~/.bashrc
Yatishs-MacBook-Air:~ Yatish$ mongod
2015-09-11T16:00:04.522-0400 E NETWORK  [initandlisten] listen(): bind() failed errno:48 Address already
in use for socket: 0.0.0.0:27017
2015-09-11T16:00:04.522-0400 E NETWORK  [initandlisten]   addr already in use
2015-09-11T16:00:04.522-0400 W -        [initandlisten] Detected unclean shutdown - /data/db/mongod.lock
is not empty.
2015-09-11T16:00:04.527-0400 I STORAGE  [initandlisten] exception in initAndListen: 98 Unable to lock fil
e: /data/db/mongod.lock errno:35 Resource temporarily unavailable. Is a mongod instance already running?,
 terminating
2015-09-11T16:00:04.528-0400 I CONTROL  [initandlisten] dbexit:  rc: 100
Yatishs-MacBook-Air:~ Yatish$
```

Fig. 14c - mongod error

To solve this issue on a mac, we need to kill the process running on port 27017. We first need to determine the process number to be able to kill it which can be done with the following command in terminal:

```
ps wuax | grep mongo
```

This will show an output like this:

```
Yatish  13035 0.5  1.8  7141128  73920   ??  S   3:59PM   0:04.01 mongod
Yatish  13071 0.4  0.0  2433796    664 s000  S+  4:09PM   0:00.01 grep mongo
```

Once we know the process number, we can simply run a kill command to kill that process and then start the mongo server again by typing mongod.

```
kill 13035
```

Neo4J

Neo4j is a type of graph database used to store highly interconnected and dynamic data. Like MongoDB and SQL databases, Neo4j provides mechanisms for querying the data, it has created specialized queries for graph data such as a relatedness index.

Neo4j community edition can be downloaded from the official site directly. neo4j[24]

Installation

Neo4J installation is very simple with only one prerequisite i.e. OpenJDK7[25]

Once you have this Java version you can directly go inside the neo4j unzipped folder and within it bin folder and type the following command to run the server:

[24] http://neo4j.com/download/other-releases/

[25] http://openjdk.java.net/

```
1 cd Downloads/neo4j-community-2.2.4/bin
2
3 neo4j start
```

```
Yatishs-MacBook-Air:bin Yatish$ neo4j start
Starting Neo4j Server...WARNING: not changing user
process [12573]... waiting for server to be ready........ OK.
http://localhost:7474/ is ready.
Yatishs-MacBook-Air:bin Yatish$
```

Fig. 14d - Neo4j connection accepted

As shown in the image, "http://localhost:7474/ is ready." The URL address lists the hostname as localhost and the port number as 7474. Copy and paste this URL into any browser to see the local page for neo4j. The first time you access the home page, you can set a new password for the neo4j server by clicking on the password link on the left pane. The default username is neo4j and the default password is neo4j. After changing the password, keep a note of the username and password. Now we can create nodes and relationships using the neo4j user interface; you can also create nodes and edges using an R program. We will create an R program that creates a graph that represents the organizational chart for company XYZ. Our nodes are employees and the edges are the different relationships that can exist between nodes, such as the hierarchical relationship 'works for' or 'part_-of', or the the binary relationships 'friends with' and 'knows' relationship, as displayed in Fig. 14d.

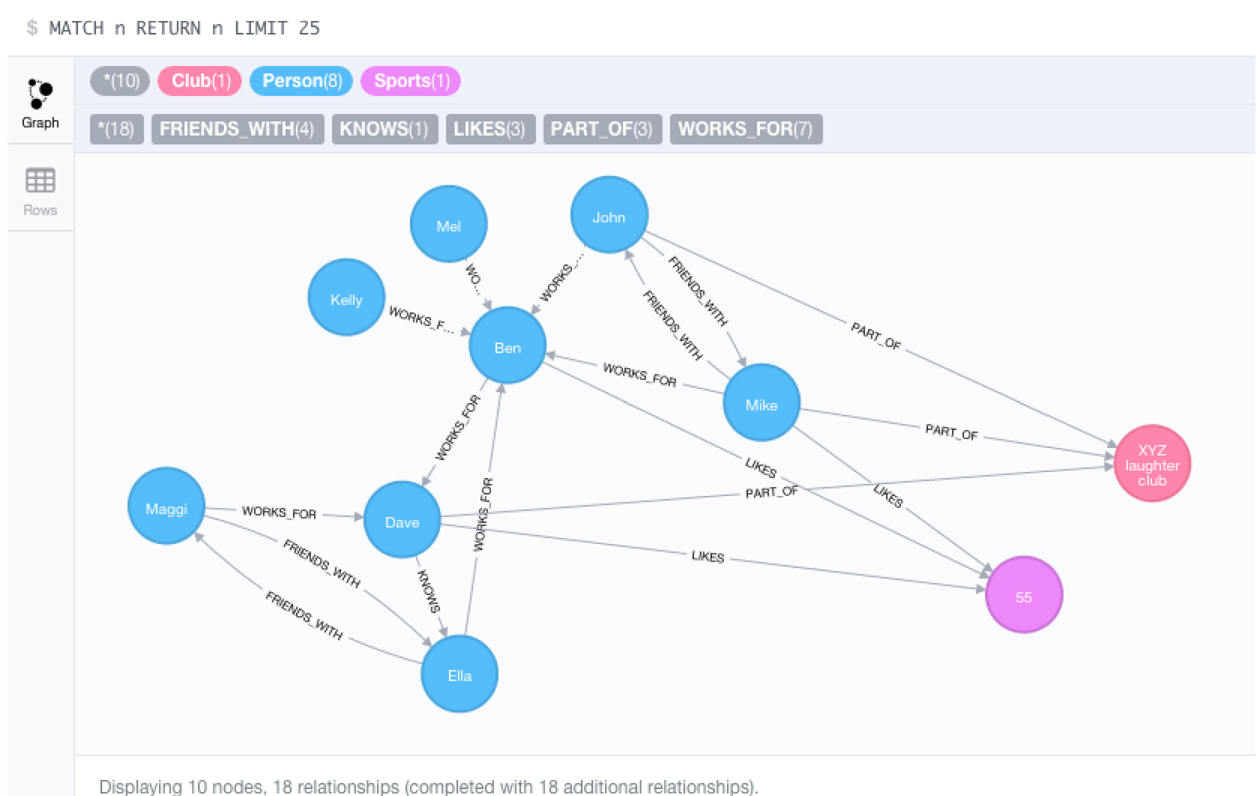


Fig. 14e - Neo4j graph

Installing neo4j in R

```
1 install.packages("devtools")
2 install.packages("httr")
3 devtools::install_github("nicolewhite/RNeo4j")
4 library(RNeo4j)
```

Establishing connection

```
1 graph = startGraph("http://localhost:7474/db/data/",username="neo4j",password="y\
2 atish")
```

Creating Nodes

The first step of creating a graph is to create nodes. Nodes are created using the createNode function of RNeo4j.

In the above graph, we have 10 nodes. We can give a specific attribute to all of these nodes which will be stored as key-value arguments.The createNode function has the following syntax:

createNode(graph object, label, key1=value1, key2=value2)

graph object is the object in which we have established a connection, the label is some label defined for this node.Labels will be later used to fetch specific data.

```
ben <-createNode(graph, "Person", name="Ben", age=45, designation="Sr. Manager")
mel <-createNode(graph, "Person", name="Mel", age=38, designation="Team Leader")
kelly <-createNode(graph, "Person", name="Kelly", age=30,
                                    designation="Programmer")
mike <- createNode(graph, "Person", name="Mike",age=27,
                                designation="Software Tester")
john <- createNode(graph, "Person", name="John", age=28,
                                designation="Android developer")
ella <- createNode(graph, "Person", name="Ella", age=28,
                                        designation="Marketing")
dave <- createNode(graph, "Person", name="Dave", age=48,
                                        designation="Zonal Head")
maggi <- createNode(graph, "Person",name="Maggi", age=28,
                                            designation="Assistant")
laugh<- createNode(graph, "Club",name="XYZ laughter club",
                                    motto="learn to laugh",members=50)
sports <- createNode(graph,"Sports",name="Cricket",host="ICC")
```

Creating relationships

Similar to the createNode() function, the createRel function is used to create a relationship between two nodes. It takes 3 or more arguments, the optional arguments provide data elements for the relationship.

createRel(node1,"Relationship label",node2, key1=value1,key2=value2)

```
r1 <- createRel(mel,"WORKS_FOR",ben)
r2 <- createRel(kelly,"WORKS_FOR",ben)
r3 <- createRel(mike,"WORKS_FOR",ben)
r4 <- createRel(john,"WORKS_FOR",ben)
r5 <- createRel(ella,"WORKS_FOR",ben)
r6 <- createRel(ben,"WORKS_FOR",dave)
r7 <- createRel(maggi, "WORKS_FOR",dave)
r8 <- createRel(maggi, "FRIENDS_WITH",ella)
r9 <- createRel(ella, "FRIENDS_WITH",maggi)
r10 <- createRel(john, "FRIENDS_WITH",mike)
r11 <- createRel(mike, "FRIENDS_WITH",john)
r12 <- createRel(dave, "PART_OF",laugh)
r13 <- createRel(john,"PART_OF",laugh)
r14 <- createRel(mike,"PART_OF",laugh)
```

```
15 r15 <- createRel(dave,"LIKES",sports)
16 r16 <- createRel(ben,"LIKES",sports)
17 r17 <- createRel(mike,"LIKES",sports)
18 r18 <- createRel(dave,"KNOWS",ella)
```

Just by creating nodes and relationships we are able to create the above graph. To get a visual representation, open the localhost link and click on the three dots on the top left corner to expand the menu bar. Then click the '*' under the node labels.

Cypher

Neo4j has its own query language called cypher. The syntax of cypher is completely different from SQL providing enough flexibility to determine relatedness and distant connections.

Let's build a simple query to fetch the details of all the employees who works for somebody.

```
1  # Query to display all the employees who works for somebody
2  > query = "
3  + MATCH (m:Person)-[:WORKS_FOR]->(works_for:Person)
4  + RETURN m.name,m.age,m.designation,works_for.name
5  + "
6  > cypher(graph, query)
7    m.name m.age     m.designation works_for.name
8  1  Maggi    28         Assistant           Dave
9  2    Ben    45       Sr. Manager           Dave
10 3    Mel    38       Team Leader            Ben
11 4  Kelly    30        Programmer            Ben
12 5   Mike    27   Software Tester            Ben
13 6   John    28 Android developer            Ben
14 7   Ella    28         Marketing            Ben
```

The MATCH keyword specifies that a query can start at a type of node and follow a specific relationship to another node. The syntax for the MATCH statement is:

Match (<this.node>)-[<with.this.relationship>]-> (<to.this.node>) and return details.

Nodes are enclosed within parenthesis () and relationships are enclosed within square brackets[]. The colon is used to create an alias for a node or a relationship. Once the aliases are defined, they are used to retrieve any corresponding attributes for the aliased node or relationship.

In the above command, we are fetching data of who in "Person" node "works for"(relationship) which "Person".

Like the SQL SELECT statement, the MATCH command also accepts a WHERE clause. Below is an example where we are limiting the returned data to people with a certain age.

```
1  > query = "
2  + MATCH (m:Person)-[:WORKS_FOR]->(works_for:Person)
3  + WHERE m.age= 28
4  + RETURN m.name,m.age,m.designation,works_for.name
5  + "
6  > cypher(graph, query)
7    m.name m.age     m.designation works_for.name
8  1  Maggi    28         Assistant           Dave
9  2   John    28 Android developer            Ben
10 3   Ella    28         Marketing            Ben
```

Combining two match queries

You may want to limit your result set to nodes that participate in more than 1 relationship. You can add additional arguments to the MATCH command to specify the additional relationships. In the example below, we are matching people who are boss/subordinate, as well as friends and the subordinate likes Sports.

```
1  > query = "
2  + MATCH (m:Person)-[:WORKS_FOR]->(w:Person),
3  + (m:Person)-[:FRIENDS_WITH]->(Person),
4  + (m:Person)-[:LIKES]->(Sports)
5  + RETURN m.name,m.age,m.designation,w.name
6  + "
7  > cypher(graph, query)
8    m.name m.age   m.designation w.name
9  1   Mike    27 Software Tester    Ben
```

Just like a SQL SELECT statement, a complex MATCH statement should be developed iteratively. Develop one MATCH relationship clause one at a time. The example below displays one example for developing the above complex MATCH statement.

```
1  > query = "
2  + MATCH (m:Person)-[:WORKS_FOR]->(w:Person),
3  + (m:Person)-[:FRIENDS_WITH]->(Person)
4  + RETURN m.name,m.age,m.designation,w.name
5  + "
6  > cypher(graph, query)
7    m.name m.age     m.designation w.name
8  1  Maggi    28         Assistant   Dave
9  2   Mike    27   Software Tester    Ben
10 3   John    28 Android developer    Ben
11 4   Ella    28         Marketing    Ben
```

The MATCH example above, returns 4 boss/subordinates who are friends. Once we are satisified with this return set, we write another MATCH query that find people who like sports. The last step is to combine all clauses and to verify the results of the intersecting result sets. Now we need to find who likes sports in a separate query and then combine the two queries together.

```
1 > query = "
2 + MATCH (m:Person)-[:WORKS_FOR]->(w:Person),
3 + (m:Person)-[:LIKES]->(Sports)
4 + RETURN m.name,m.age,m.designation,w.name
5 + "
6 > cypher(graph, query)
7   m.name m.age   m.designation w.name
8 1    Ben    45     Sr. Manager   Dave
9 2   Mike    27 Software Tester    Ben
```

If a particular query does not return a result set that means that there is no matching result for that condition in your graph. For example:

```
1 > query = "
2 + MATCH (m:Person)-[:WORKS_FOR]->(w:Person),
3 + (m:Person)-[:FRIENDS_WITH]->(w:Person)
4 + RETURN m.name,m.age,m.designation,w.name
5 + "
6 > cypher(graph, query)
7 #returns nothing as there is nobody who is friends with either Dave or Ben (w:Pe\
8 rson here is an alias for Employers)
```

cyperToList

The cypherToList function converts the result of a query to a List. The collect function creates a set for the matching values. Here we are creating a result set with all fields from Person node and a collection field that contains all the employees that report to a person.

```
> query = "
+ MATCH (m:Person)-[:WORKS_FOR]->(w:Person)
+ RETURN w, Collect(m.name) AS employees
+ "
> cypherToList(graph, query)
[[1]]
[[1]]$w
< Node Object >
$name
[1] "Ben"

$age
[1] 45

$designation
[1] "Sr. Manager"


[[1]]$employees
[[1]]$employees[[1]]
[1] "Mel"

[[1]]$employees[[2]]
[1] "Kelly"

[[1]]$employees[[3]]
[1] "Mike"

[[1]]$employees[[4]]
[1] "John"

[[1]]$employees[[5]]
[1] "Ella"

[[2]]
[[2]]$w
< Node Object >
$name
[1] "Dave"

$age
[1] 48
```

```
43
44 $designation
45 [1] "Zonal Head"
46
47
48 [[2]]$employees
49 [[2]]$employees[[1]]
50 [1] "Ben"
51
52 [[2]]$employees[[2]]
53 [1] "Maggi"
```

In the above example, we are capturing the two employers and collecting the name of people who works for them under the alias called employees.

```
1 > emp<-cypherToList(graph, query)
2 > emp[[2]]$employees
3 [[1]]
4 [1] "Ben"
5
6 [[2]]
7 [1] "Maggi"
```

Parameterized queries

Typically when writing code, you will want to write code that accepts arguments. This way you can use the same code when the values you want to match changes. This is known as parametrizing code. We can parametrize the MATCH command by providing variables that contain the values we want to match. Within the MATCH command all variables are surrounded by curly braces {}. This signals that the variable needs to be evaluated to determine the value that should be included in the MATCH statement. Below is an example of a parametrized query.

```
1  > query = "
2  + MATCH (m:Person)-[:WORKS_FOR]->(works_for:Person)
3  + WHERE m.age={age}
4  + RETURN m.name,m.age,m.designation,works_for.name
5  + "
6  > cypher(graph, query,age=28)
7    m.name m.age     m.designation works_for.name
8  1  Maggi    28         Assistant           Dave
9  2   John    28 Android developer            Ben
10 3   Ella    28         Marketing            Ben
```

```
> query = "
+ MATCH (m:Person)-[:WORKS_FOR]->(w:Person)
+ Where w.name= {name}
+ RETURN w, Collect(m.name) AS employees
+ "
> cypherToList(graph, query,name="Dave")
[[1]]
[[1]]$w
< Node Object >
$name
[1] "Dave"

$age
[1] 48

$designation
[1] "Zonal Head"


[[1]]$employees
[[1]]$employees[[1]]
[1] "Ben"

[[1]]$employees[[2]]
[1] "Maggi"
```

Finding the shortest path

Within a graph, there may be multiple paths from one node to another node. We can use the shortestPath function to calculate the shortest path from one node to another node. When traversing the graph, we can specify the relationship to use as a edge connecting the nodes. In the example below we limit paths from kelly to dave using only edges that represent the WORKS_FOR relationship. This MATCH query will return the middle managers between the two people. In our example kelly works for Ben and Ben works for Dave, hence the shortest path is 2.

```
query = "
MATCH p = shortestPath((kelly:Person)-[:WORKS_FOR*]->(dave:Person))
WHERE kelly.name = 'Kelly' AND dave.name = 'Dave'
RETURN p
"

p = cypherToList(graph, query)[[1]]
p$p$length
```

Visualizing graphs in R

The library igraph can be used to display graphs. We need to specify the layout of the graph, we use the betweeness and closeness measure to display the graph.

```
library(igraph)
> query="
+ MATCH (n) -->(m)
+ RETURN n.name,m.name
+ "
> cypher(graph, query)
   n.name            m.name
1    Dave           Cricket
2    Dave XYZ laughter club
3    Dave              Ella
4   Maggi              Dave
5   Maggi              Ella
6     Ben           Cricket
7     Ben              Dave
8     Mel               Ben
9   Kelly               Ben
10   Mike           Cricket
11   Mike               Ben
12   Mike              John
13   Mike XYZ laughter club
14   John               Ben
15   John              Mike
16   John XYZ laughter club
17   Ella               Ben
18   Ella             Maggi
```

```
1  > edgelist = cypher(graph, query)
2  > ig = graph.data.frame(edgelist, directed=F)
3  > betweenness(ig)
4                 Dave              Maggi                Ben                Mel
5            7.1000000          0.0000000         20.0166667          0.0000000
6                Kelly               Mike               John               Ella
7            0.0000000          2.1666667          1.0000000          2.9000000
8              Cricket XYZ laughter club
9            0.5333333          1.2833333
10 > closeness(ig)
11                Dave              Maggi                Ben                Mel
12          0.07692308         0.05000000         0.09090909         0.05263158
13               Kelly               Mike               John               Ella
14          0.05263158         0.06666667         0.06250000         0.06666667
15             Cricket XYZ laughter club
16          0.06666667         0.05882353
```

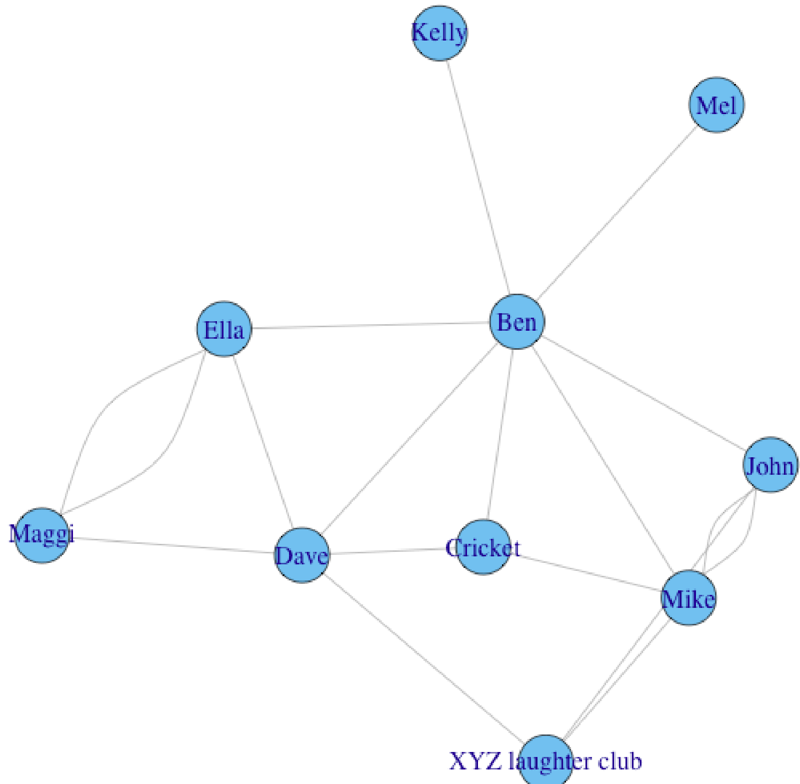


Fig. 14f - Plotting via R

Another package called network provides a different visualization for the graphs. Other packages to review are “sna” , “GGally” and “intergraph”.

```
1  install.packages("intergraph")
2  install.packages("sna")
3  install.packages("network")
4  install.packages("GGally")
5  library(network)
6  library(GGally)
7
8  net = network(edgelist)
9  ggnet(net, label.nodes=TRUE,color="dark blue",segment.color="lightgrey",alpha=0.\
10 5)
```

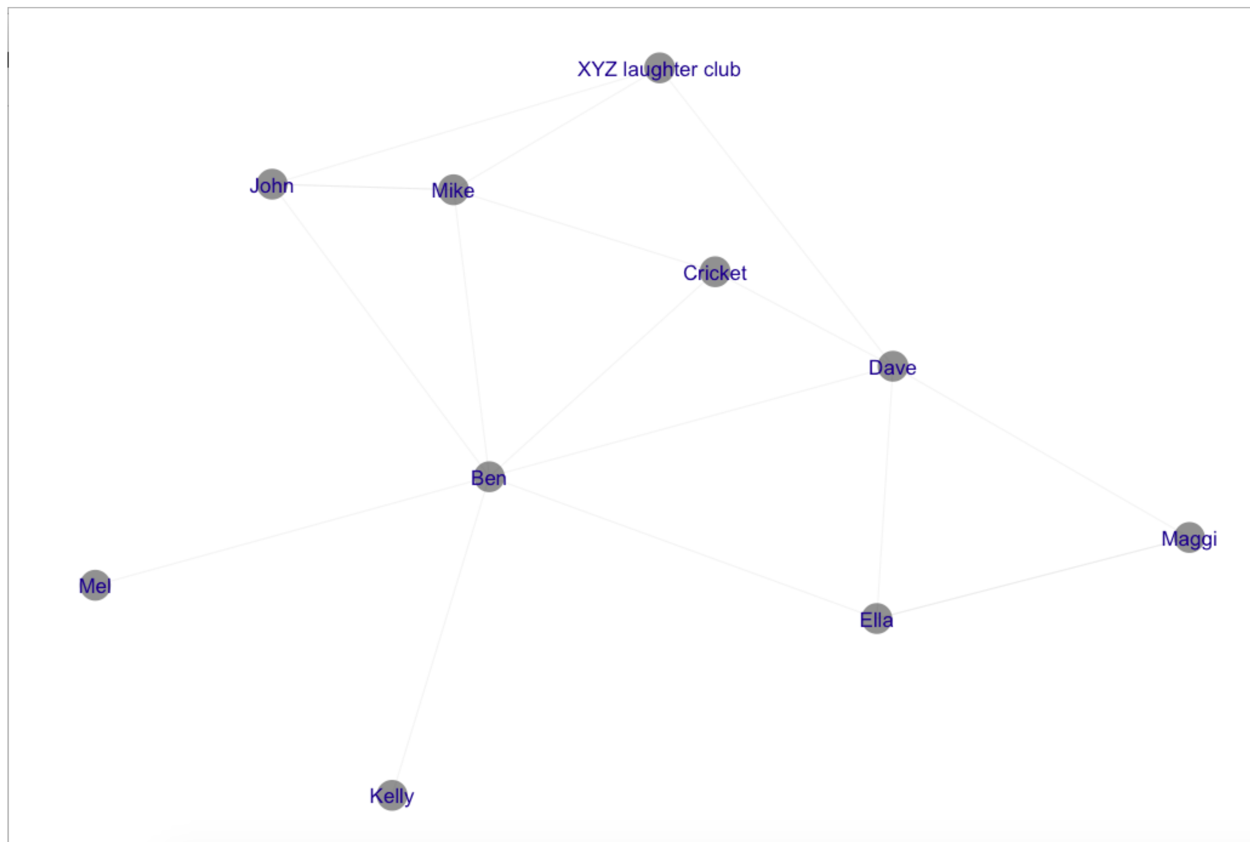


Fig. 14g - Plotting via R

Chapter 15

Data Analysis

Data analysis is the process of evaluating the clean and transformed data using analytical and logical reasoning to examine each component of the data. When analyzing data it is critical to ask where the data comes from and how it was produced, obtained or collected. In particulas, the quality of the data must be assessed before analysis. In any organization, processes that ensure accurate collection and curation of data are described under data provenance.

Data analysis process is aptly summarized in figure 15a.

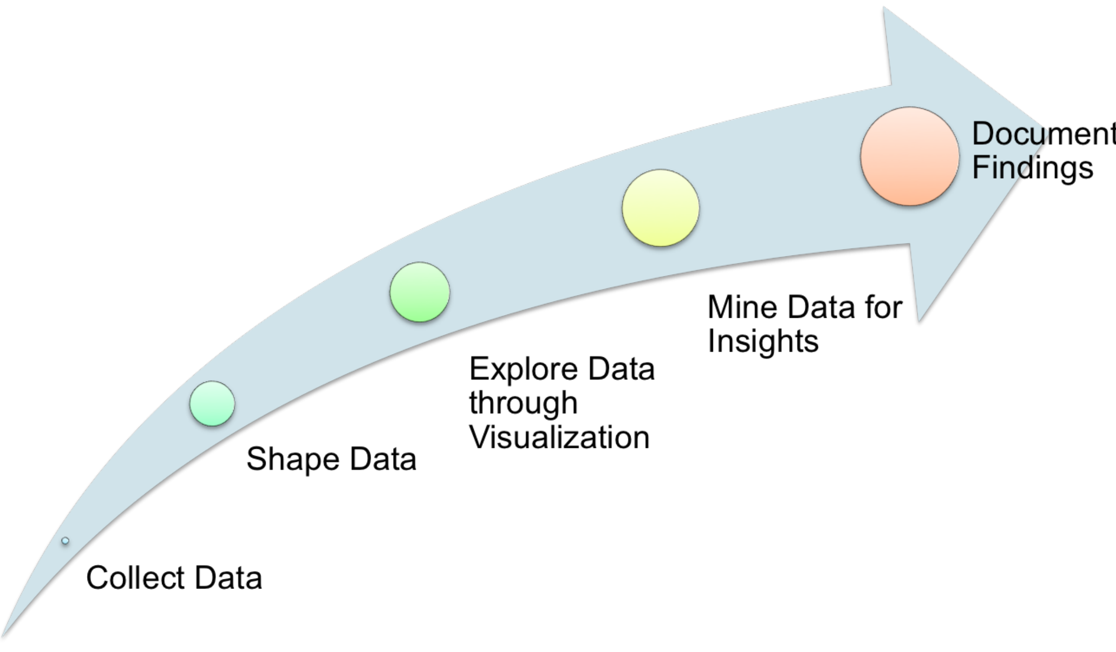


Fig. 15a - Data analysis process

Exploring data through visualization

Categorical data

Categorical data can be summarized using tables and graphs. A summary table displays the frequency, value or percentage for analysis. The summary function in R is used to get the summary of given vector or dataframe. Table function can be used to find the frequency of data points.

```
1  > clinical.trial <-
2  +   data.frame(patient = 1:100,
3  +   age = rnorm(100, mean = 60, sd = 6),
4  +   center = sample(paste("Center", LETTERS[1:5]), 100, replace = TRUE))
5
6  > summary(clinical.trial)
7      patient              age                center
8   Min.   :  1.00   Min.   :48.11   Center A:20
9   1st Qu.: 25.75   1st Qu.:55.50   Center B:22
10  Median : 50.50   Median :59.79   Center C:17
11  Mean   : 50.50   Mean   :60.09   Center D:19
12  3rd Qu.: 75.25   3rd Qu.:63.90   Center E:22
13  Max.   :100.00   Max.   :82.18
14
15 > table(clinical.trial$center)
16
17 Center A Center B Center C Center D Center E
18       20       22       17       19       22
```

Categorical data can be represented using bar charts or pie charts. A bar chart shows each category's frequency or percentage as the length of the bar. A pie chart shows the allocation of each category.

```
1  barplot(data,main="Clinical trials",xlab="centers",ylab="number of centers")
```

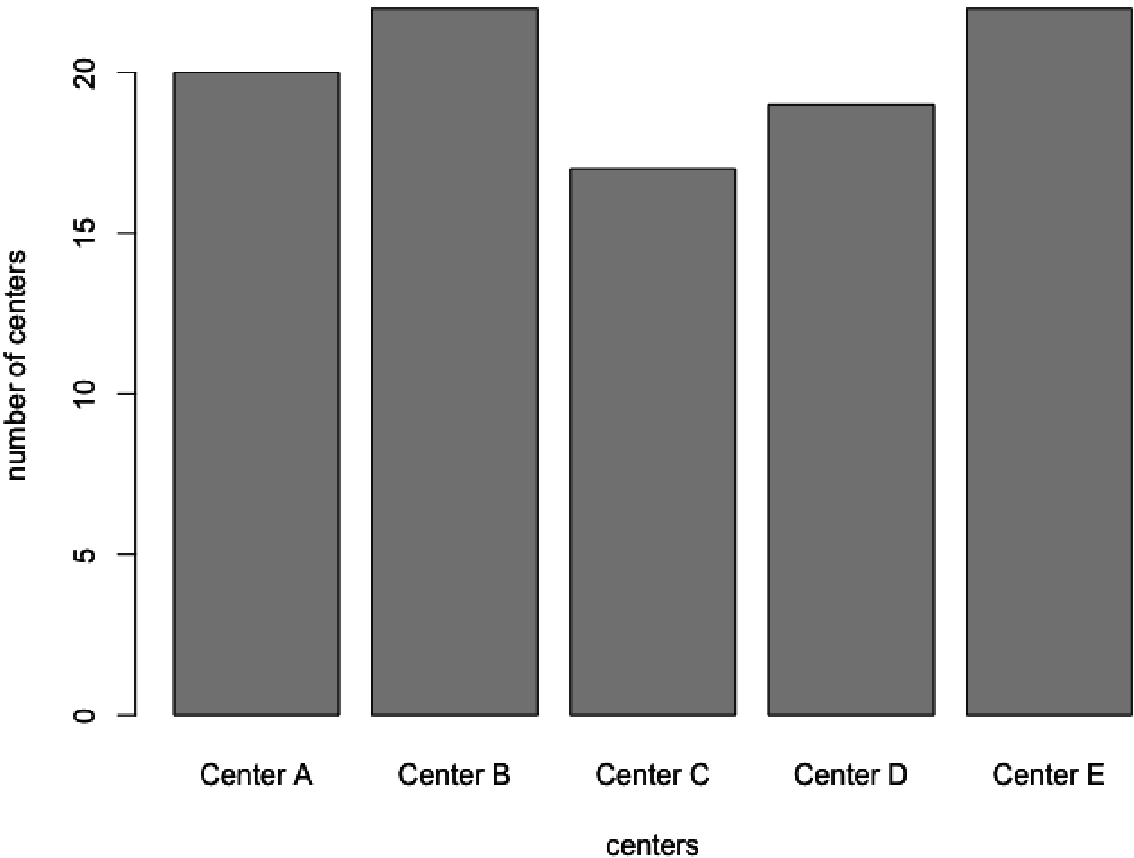


Fig. 15b - Bar chart

```
1  data<-table(clinical.trial$center)
2  lbls <- paste(names(data), "\n", data, sep="")
3  pie(data,labels=lbls,main="Clinical trials")
```

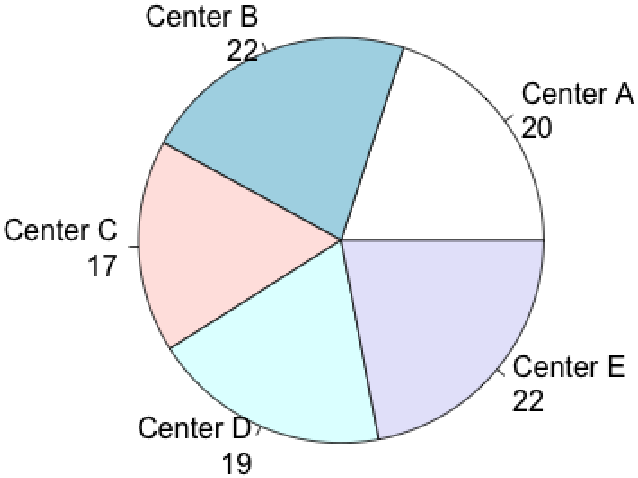


Fig. 15c - Pie chart

Numerical data

Visualization of numerical data can be done using different set of graphs: Histogram, Scatter plot, Time series plot.

A histogram shows the frequency distribution in a bar graph format. Histograms are useful in detecting the probability distribution function (cdf).

```
1 hist(AirPassengers)
```

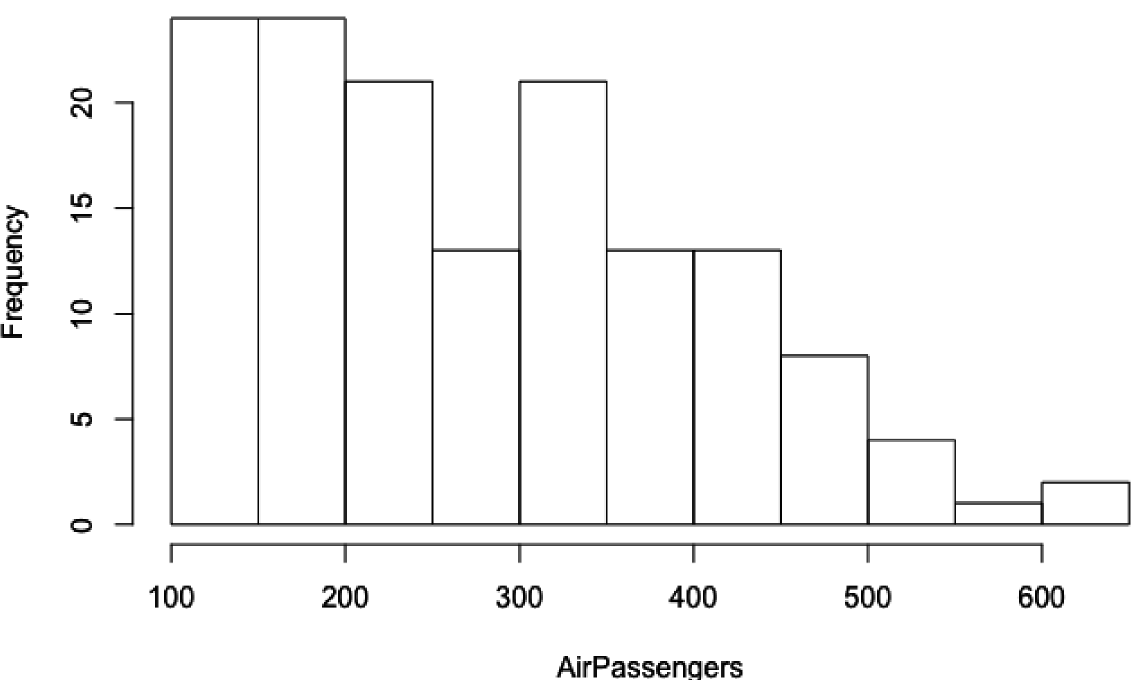


Fig. 15d - Histogram

Scatter plots helps visualize paired observations. It is useful for detecting correlations within data.

```
plot(cars)
# cars is the built-in dataset in R
```

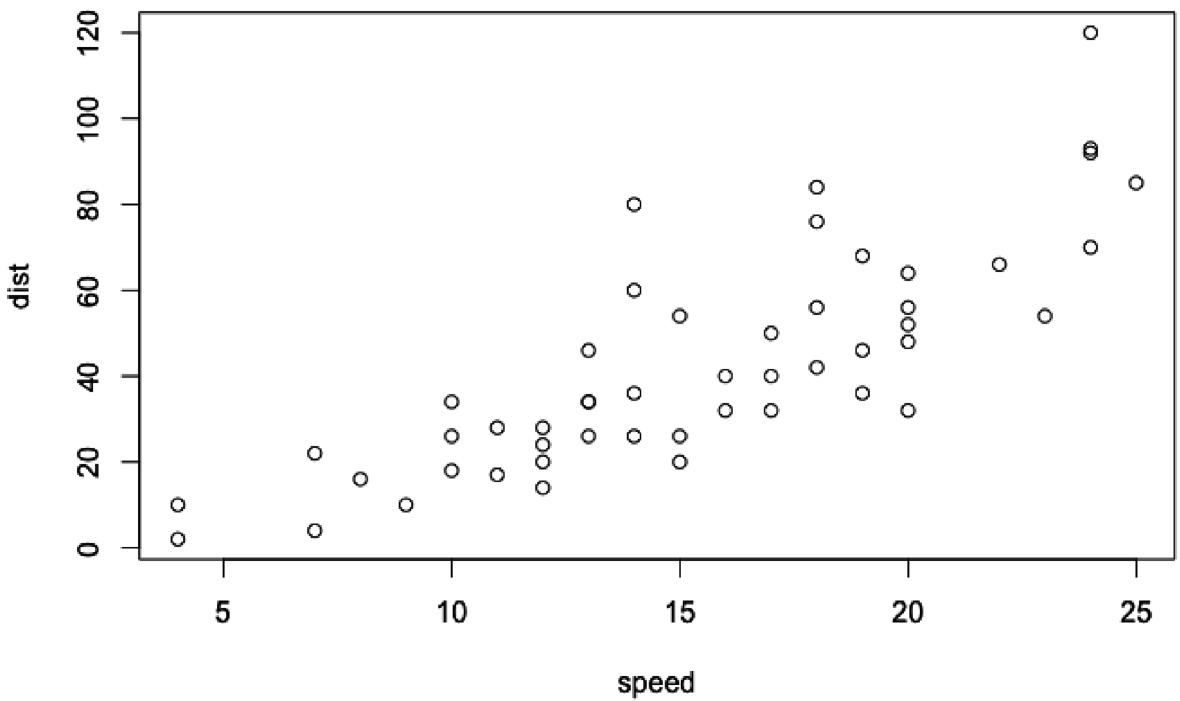


Fig. 15e - Scatter plot

A time series plot shows the pattern of change over time.

```
1   > Lines <- "Date          Visits
2   + 10/1/2010   696537
3   + 10/2/2010   718748
4   + 10/3/2010   799355
5   + 10/4/2010   805800
6   + 10/5/2010   701262
7   + 10/6/2010   531579
8   + 10/7/2010   690068
9   + 10/8/2010   756947
10  + 10/9/2010   718757
11  + 10/10/2010  701768
12  + 10/11/2010  820113
13  + 10/12/2010  645259"
14
15  # reading the table
16  > dm <- read.table(text = Lines, header = TRUE)
```

```
17  > dm$Date <- as.Date(dm$Date, "%m/%d/%Y")
18
19  #Plotting Visits vs Date
20  > plot(Visits ~ Date, dm, xaxt = "n", type = "l")
21  #xaxt help customized x axis using axis function.
22  > axis(1, dm$Date, format(dm$Date, "%b %d"), cex.axis = .7)
```

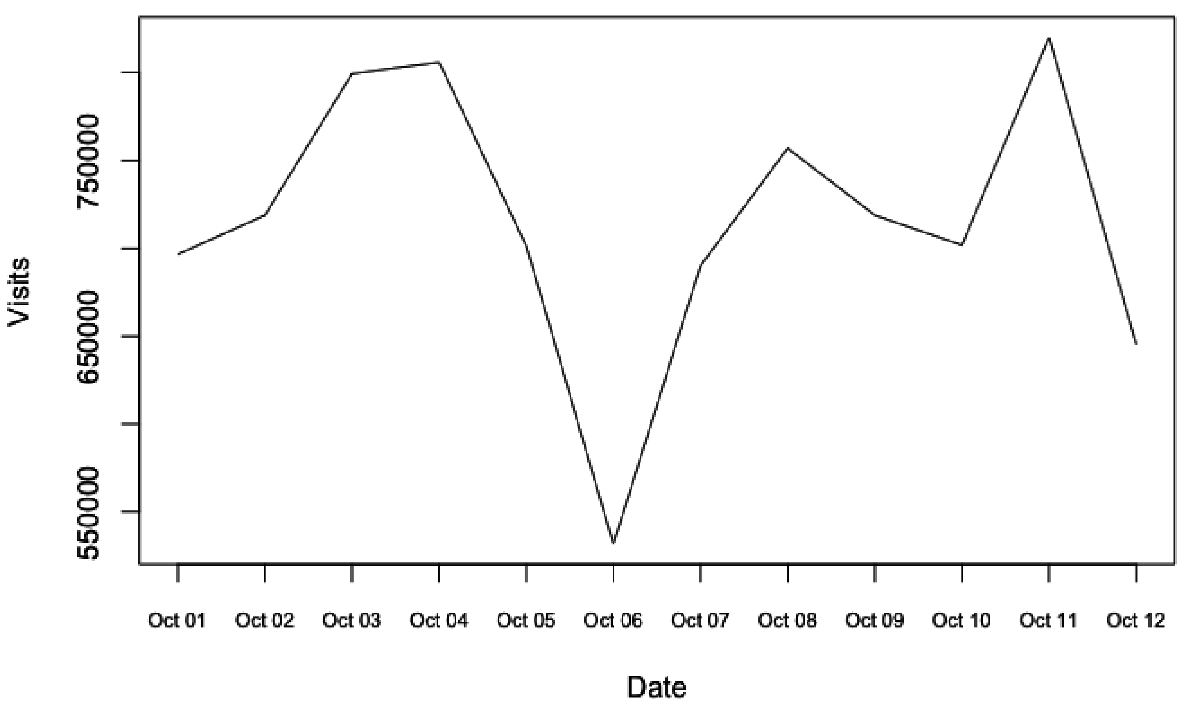


Fig. 15f - Time series plot

Chapter 16

Characterizing data through statistics

Statistics is a branch of mathematics that analyzes and transforms numeric data into useful information for decision making and prediction.Statistics helps quantify uncertainty and aids in rational prediction.

Statistics is broadly organized into descriptive and inferential methods:

1. **Descriptive methods** describe the properties of a data set, such as the mean (average) or the maximum.
2. **Inferential methods** draw general conclusion from small samples and compare central tendencies in multiple data sets.

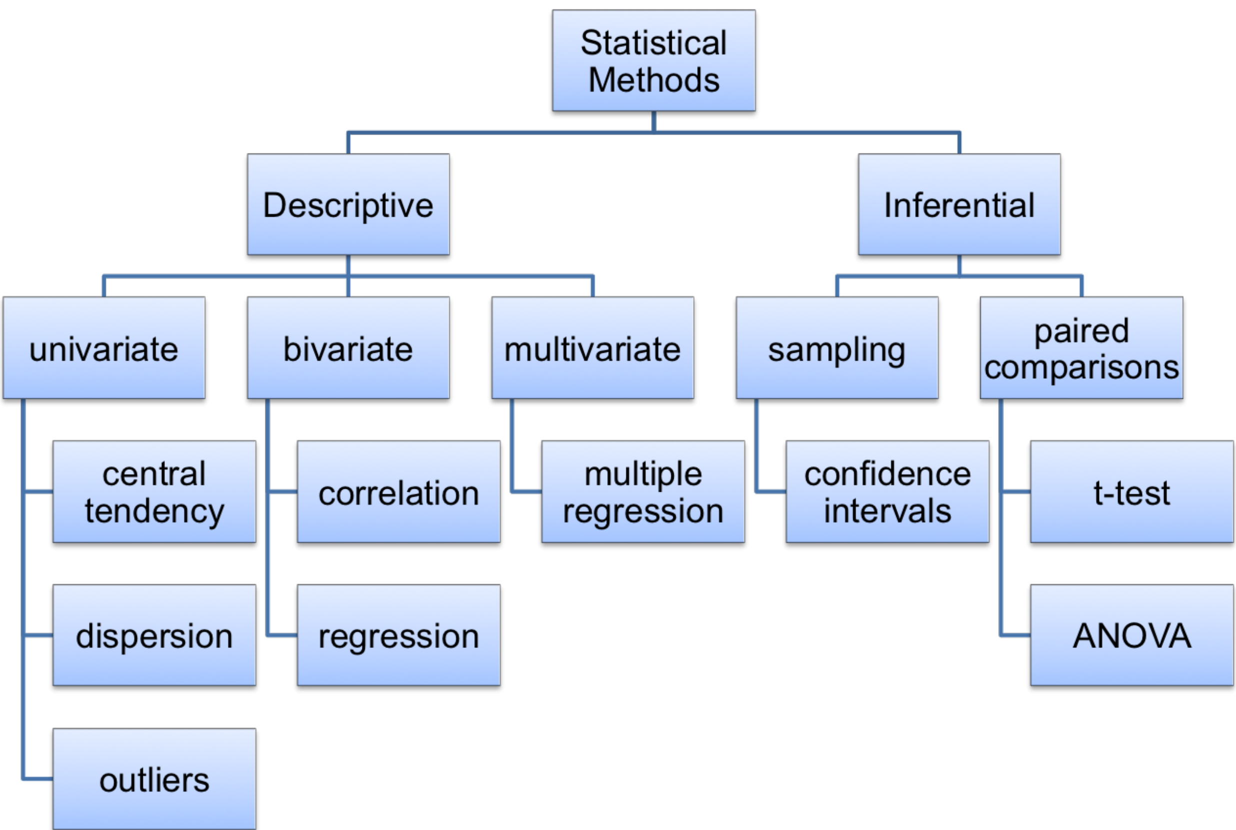


Fig. 16a - Taxonomy of statistical methods

Important terminologies related to statistics

1. **Variable**: a characteristic of an item or individual, e.g. salary, age, income, weight, experience.
2. **Data**: The different values associated with a variable, e.g. 168lbs, 203lbs.
3. **Population**: A large group that is to be measured.
4. **Sample**: A subset of a population that will be measured.
5. **Parameter**: A numerical measure that describes the characteristics of a population, e.g. mean.

Descriptive Statistics

Descriptive statistics describes the central tendency i.e. the extent to which the data values group around a central value and variation i.e. the amount of dispersion or scattering of the data values.

There are three major measures of central tendency.

1. **Mean**: it is the average values of a set of data values.
2. **Median**: it is the middle value in an ordered list of values. If the number of values is even, then the median is the average of the two middle values.
3. **Mode**: it is the value that occurs most frequently in the data set. The mode, unlike the mean, is not affected by outliers.

In R we can use built-in functions to find mean and median but there is no straight-forward way to find mode.

```
1  > x <- c(12,7,3,4.2,18,2,54,-21,8,-5,3)
2
3  > mean(x)
4  [1] 7.745455
5
6  > median(x)
7  [1] 4.2
8
9  > names(sort(-table(x)))[1] #Using the negative table function to find
10 #frequency and then sorting and picking up the first value.
11 [1] "3"
```

Variation can be measured by following measures.

1. **Range**: it is the difference between the smallest and the largest value.
2. **Variance**: it is the average of the squared deviations of the values from the mean.
3. **Standard deviation**: it is the square root of the variance and is the most commonly used measure for dispersion around the mean. A smaller standard deviation indicates that the data is more closely clustered around the mean, while a larger value implies more spread.

```
1  > x <- c(12,7,3,4.2,18,2,54,-21,8,-5,3)
2  > max(x)
3  [1] 54
4  > min(x)
5  [1] -21
6  > var(x)
7  [1] 334.2727
8  > sd(x)
9  [1] 18.28313
```

Outliers

Outliers are the extreme values that can skew the analysis of data. Values that are more than 3 standard deviations from the mean are generally considered outliers. Z-score is used to locate outliers. The z-score is the number of standard deviations a data value is from the mean. Outliers have a z-score of Ã, Â± 3.0.

```
1  > x <- c(12,7,3,4.2,18,2,54,-21,8,-5,3)
2  > scale(x)
3               [,1]
4   [1,]  0.23270338
5   [2,] -0.04077281
6   [3,] -0.25955377
7   [4,] -0.19391949
8   [5,]  0.56087482
9   [6,] -0.31424901
10  [7,]  2.52990344
11  [8,] -1.57223952
12  [9,]  0.01392242
13 [10,] -0.69711569
14 [11,] -0.25955377
15 attr(,"scaled:center")
16 [1] 7.745455
17 attr(,"scaled:scale")
18 [1] 18.28313
```

Inferential statistics

Statistical inference combines the methods of descriptive statistics with the theory of probability to infer characteristics of a large population from a small sample. The sample must be selected randomly and must be "large enough" to be statistically significant. Statistical significance means that the characteristics of the sample are likely not due to chance or random error. In most sciences,

a confidence level of 95% is generally accepted as statistically significant, i.e., we accept a 5% probability of being wrong about our conclusion.

A sample is a small set of randomly selected representatives from a larger population. When it is impossible (or very difficult) to measure a population, then select a sample and extrapolate results for the population.

Standard error

The standard error of a sample is a measure of the sampling error and can be used to estimate the standard deviation of a population. To find the standard error in R, "plotrix" package can be installed to use the "std.error" function.

```
1 install.packages("plotrix")
2 library("plotrix")
3 x <- c(12,7,3,4.2,18,2,54,-21,8,-5,3)
4 std.error(x)
```

Random sample

Samples must be drawn randomly from population, which means that each element of the population has the same probability of being included in the sample. Random samples are often drawn through random events, such as assigning numeric identifiers to each element and selecting a set of random numbers through a computer or a random number table.

sampling can be done in R using the function "sample()"

```
1 > sample(1:10)
2  [1]  8  3 10  6  2  9  4  1  5  7
3
4 > sample(1:100, size=10)
5  [1] 26  2 35 55 34 29 25 37 96 77
6
7 > sample(1:100, size=10, replace=T)
8  [1] 78 98 16 32  5 46 11  1 11 15
```

The sample function can be used to randomly select a small size sample from population. The replace argument of sample function allows duplicate entities to be selected from population if turned on.

Correlation

A correlation is a relationship between two variables in which one variable changes in a quantifiable way with another.

For example, there are correlations between: * Weight and cholesterol level * Task completion time and task complexity * Cursor positioning time and distance to target

The strength of a correlation is measured by the coefficient of correlation “R”. The value of R ranges from -1 to +1.

- Positive: as one variable increases, the other increases as well
- Negative: as one variable increases, the other decreases

An absolute value close to 1 is a strong correlation, whereas a value close to 0 indicates little to no correlation.

In R, correlation can be find using function “cor()”.

```
1  > data<- "Sample  IQ  Spelling
2  + 1 115 34
3  + 2 87  18
4  + 3 104 28
5  + 4 121 26
6  + 5 96  19
7  + 6 99  20
8  + 7 136 26"
9
10 > dm <- read.table(text =data, header = TRUE)
11
12 > cor(dm$IQ,dm$Spelling)
13 [1] 0.6159423
```

Once a correlation has been established, the relationship can be mathematically quantified in a formula through regression analysis which will be discussed further with forecasting.

Forecasting Trends

Forecasting refers to the process of using statistical procedures to predict future values of a time series based on historical trends. Forecasting of demand, storage growth, resources, network traffic, orders, and so forth is a key responsibility of an information scientist.

Forecasting Strategies:

1. Qualitative models
2. Time-Series models
3. Causal models

Qualitative models incorporate expert opinions and subjective factors. These models are useful when subjective factors are thought to be important or when accurate quantitative data is difficult to obtain.

Common qualitative techniques are:

- Delphi (single and wideband)
- Expert Judgment
- Bottom-Up Composite
- Stakeholder Survey

Time-series models attempt to predict growth based on historical data. Common time-series models are:

- Moving average
- Exponential smoothing
- Trend projections

R has a forecast package for moving average, exponential smoothing and trend projections.

Trend projections fits a trend curve to a series of historical data points. The curve is projected into the future for medium to long range forecasts. In R, it is very easy to plot trend projections with "decompose()" function on any time series data vector.

```
1  > class(AirPassengers)
2  [1] "ts" # time series class
3
4  > data<-decompose(AirPassengers)
5
6  > plot(data)
```

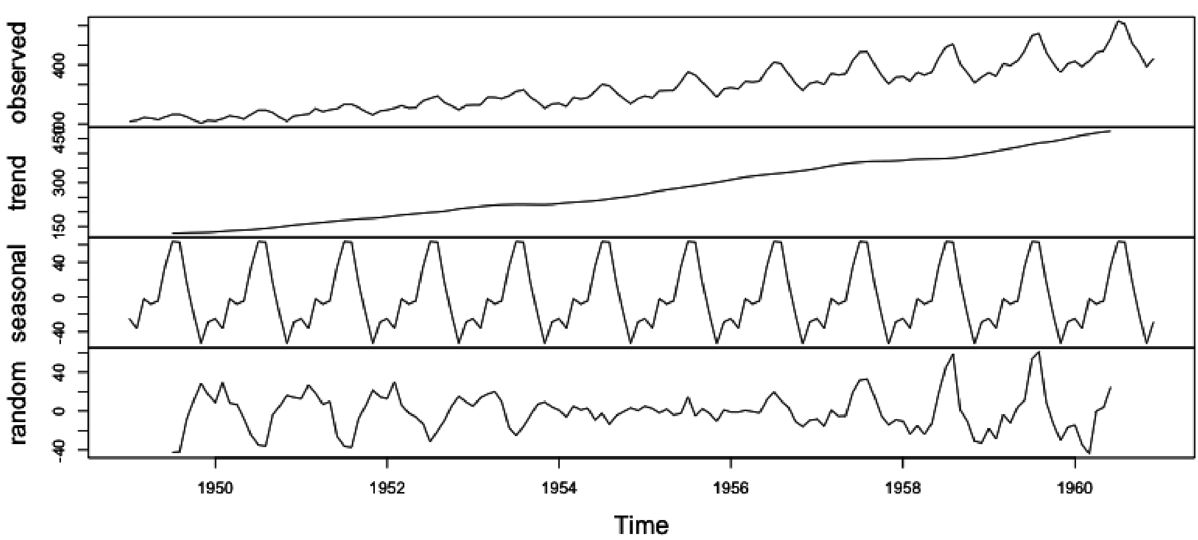


Fig. 16b - Trend projection

Moving average estimates future values at time t by averaging values of the time series within k periods of t. This method works best when the data does not contain any trend or cyclic patterns. In R, "arima()" function (ARIthmetic Moving Average) of forecast package is used for predicting future values.

```
1  > fit <- arima(AirPassengers, order=c(1,0,0), list(order=c(2,1,0), period=12))
2  #Creating a fit variable according to the dataset
3  > fore <- predict(fit, n.ahead=24)
4  #using the predict function on fitted values to predict future values.
5  #n.ahead is the number of steps ahead for which prediction is required
6
7  > # error bounds at 95% confidence level
8  > U <- fore$pred + 2*fore$se #upper confidence level
9  > L <- fore$pred - 2*fore$se #lower confidence level
10
11 > ts.plot(AirPassengers, fore$pred, U, L, col=c(1,2,4,4), lty = c(1,1,2,2))
12 #plotting time series data with legend
13 > legend("topleft", c("Actual", "Forecast", "Error Bounds (95% Confidence)"), co\
14 l=c(1,2,4), lty=c(1,1,2))
```

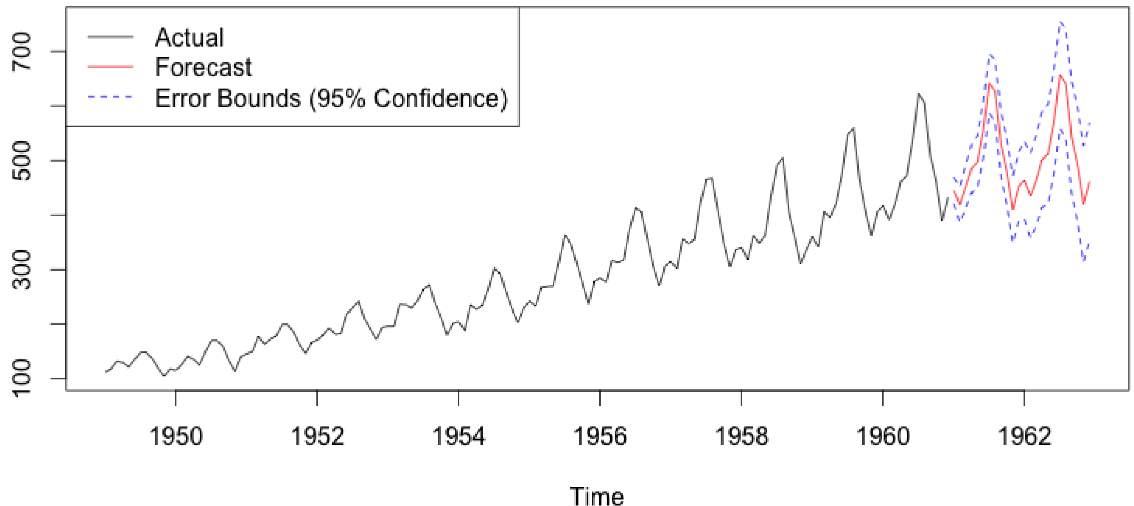


Fig. 16c - Forecast using moving average method

Exponential smoothing is good when the data has no trend or seasonal patterns. Unlike a moving average, this technique gives greater weight to the most recent observations of the time series. In R, we use "ses()" function for simple exponential smoothing and "holt()" for holt-winters smoothing.

```
1  > data<-AirPassengers
2  > exp <- ses(data)
3  > plot(exp)
```

Fig. 16d - Forecast using simple exponential smoothing method

Causal models take other factors into account and are often more accurate. The objective is to build a model with the best statistical relationship between the variable being forecast and the independent variables. Regression analysis is the most common technique used in causal modeling.

Regression analysis is used to understand the relationship between variables and predict the value of one variable based on another variable. Simple linear regression models have only two variables whereas multiple regression models have more than two variables.

Regression attempts to fit a curve to a series of historical data points which is then used to make predictions. The simplest is a linear (straight line) model developed using simple linear regression analysis. Regression models are a mathematical equation used to predict a value based on empirical observations. The prediction is never correct, but depending on the "fit of data" can be reasonably good.

The variable to be predicted is called the dependent variable or the response variable. The value of this variable depends on the value(s) of the independent variable(s) or the explanatory or the predictor variable. Multiple regression models have several independent variables.

For regression to be useful, a correlation must exist between the independent and the dependent variable. Correlation quantifies how well one variable's values move in accordance with changes in the other variable. Regression is an equation that mathematically captures how one variable changes with the other.

The fit of the regression line is measured by the coefficient of determination (R^2). The closer R^2 is to 1 the better the regression model fit and the more accurate the prediction.

Note- R^2 is one part of measuring the "quality" of a regression model: the other is

statistical significance.

After the correlation between two variables are established and visualized using scatter plot, relationship is mathmatically quantified in a formula using regression. The “lm()” function is used in R to fit a linear model where the response variable is on the left separated by “∼” from the explanatory variable. This linear model is saved as an object, named “regression.fit” in our example in R log session. Using a summary() function on this linear model object gives us a lot of information like intercept, slope of relationship, standard errors, R^2 etc.

```
1  > regression.fit<-lm(speed~dist,data=cars)
2
3  plot(speed ~ dist, data = cars,
4         xlab = "Stopping distance (ft)",
5         ylab = "Speed (mph)",
6         main = "Speed and Stopping Distances of Cars"
7  )
8
9  abline(regression.fit,col="red")
```

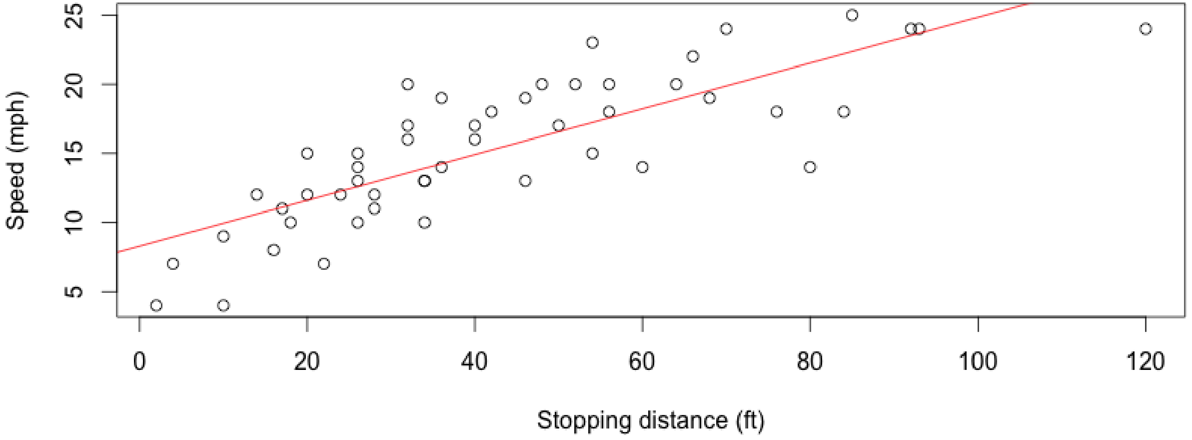


Fig. 16e - Regression plot

```
> summary(regression.fit)

Call:
lm(formula = speed ~ dist, data = cars)

Residuals:
    Min      1Q  Median      3Q     Max
-7.5293 -2.1550  0.3615  2.4377  6.4179

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  8.28391    0.87438   9.474 1.44e-12 ***
dist         0.16557    0.01749   9.464 1.49e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.156 on 48 degrees of freedom
Multiple R-squared:  0.6511,	Adjusted R-squared:  0.6438
F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Plotting this linear model object gives a set of 4 graphs namely: Residuals vs Fitted graph, Normal Q-Q plot, Scale-Location graph and Residuals vs Leverage graph.

```
par(mfrow=c(2,2))
plot(regression.fit)
```

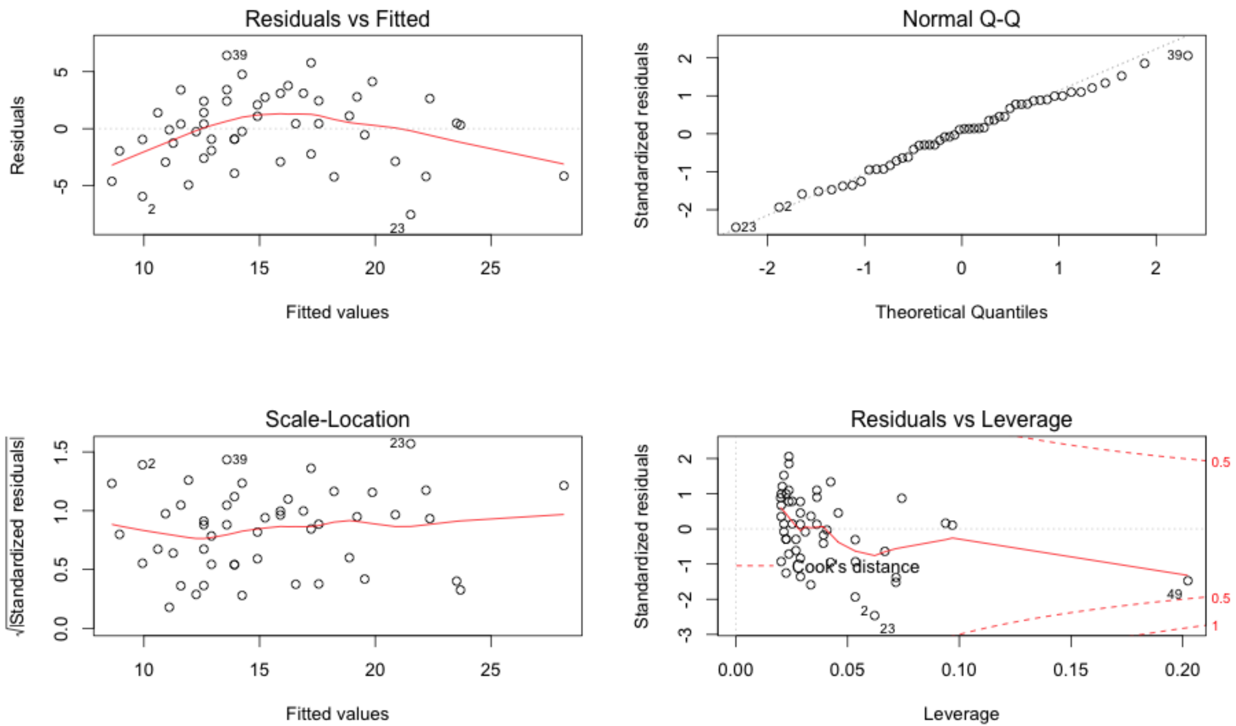


Fig. 16f - Linear model regression plots

In the first plot of residual errors vs fitted values, points that tend towards being outliers are labeled. If any pattern is apparent in the points on this plot, then the linear model may not be the appropriate one. Second and third graph of Q-Q plot and Scale-Location plot helps us determine whether our residual errors are normally distributed or not. Last plot of Residuals vs Leverage helps us identify the cases having undue influence on the regression relationship and needs further investigation.

Similar to linear regression, multiple regression analysis is also very straight-forward in R. With the "lm()" function, one can add the independent variables separated by "+" sign. For example in an imaginary dataset of speed with distance and time regression analysis can be done as shown in R log session below.

```
1  > regression.fit<-lm(speed~dist+time,data=imaginaryDataset)
```

Chapter 17

Validating data through hypothesis testing

Once data collection is completed and basic decriptive statistics functions are performed on dataset, we use the appropriate statistical test to see if there's a statistically significant different between the performance or error rate between the groups.

The first step of performing any statistical testing is to determine if the data is normally distributed or not. One way to find out the normal distribution is by plotting Q-Q plot as discussed in chapter 16. To determine the normal distribution mathematically instead of visually we use Shapiro-wilk normality test. Based on whether the data is normally distributed or not, appropriate statistical testing can be done.

- Student's t-test if the data is normally distributed.
- Kruskal-Wallis or Mann-Witney test of the data is not normally distributed.

Hypothesis

A hypothesis is a research statement. There are two types of hypothesis:

- Null hypothesis
- Alternate hypothesis

Null hypothesis states that there is no significant difference between specified populations, any observed difference being due to sampling or experimental error. Statistical tests determine the probability that the null hypothesis is not true, which means that we must accept the alternate hypothesis. The accepted probability is generally 0.05. So if $p < 0.05$ the difference is statistically significant.

To test if a hypothesis is true, following steps are taken:

- State the null hypothesis
- Perform an appropriate statistical test
- Evaluate the probability that the null hypothesis is true and either accept it or reject it

There are different types of statistical tests. All the statistical tests cannot be covered in this book.

- "t-test" tests if the mean between two samples is different.
- "Chi-squared test" tests if proportions of a binary variable between two samples are different.
- "ANOVA" tests if two or more factors influence the difference between two samples

p-value

The p-value characterizes the probability that the null hypothesis is true. If this value is less than 5% (0.05), then we conclude that the null hypothesis is too unlikely to be true, which means that the alternate hypothesis must be true. The p-value level of 0.05 is an agreed upon level and differs between different research domains.

Example Analysis

A user experience design team is trying to decide which sales order web page is more effective. They have collected data through focus groups that asked groups of users to attempt an ordering use case for both interfaces. Each invited user performed both tasks. The data is reported in matching pairs, i.e., each subject first performed the task using version A of the design, then version B. To reduce the likelihood of task dependency, subjects should be given one design at random before the other design.

```
1  > data<-"VersionA VersionB
2  + 22  22
3  + 24  34
4  + 34  45
5  + 18  43
6  + 22  31
7  + 39  38
8  + 32  45
9  + 28  32
10 + 31  38
11 + 41  29
12 + 38  41
13 + 29  58"
14 > dm <- read.table(text =data, header = TRUE)
15 > dm
16    VersionA VersionB
17 1        22       22
18 2        24       34
19 3        34       45
20 4        18       43
21 5        22       31
22 6        39       38
23 7        32       45
24 8        28       32
25 9        31       38
26 10       41       29
27 11       38       41
```

```
28 12       29       58
29
30 > mean(dm$VersionA)
31 [1] 29.83333
32 > mean(dm$VersionB)
33 [1] 38
```

Null Hypothesis: "There is no difference in average task completion time between the two page designs."

Testing approach: We need to evaluate the difference between means of two samples, so a t-test or a one-way ANOVA are useful. We must first check if the data is reasonably normally distributed in order to use these tests. If they are not, then a non-parametric test, such as a Kruskal-Wallis test must be used.

Step 1: Perform Shapiro-wilk normality test.

In Shapiro-wilk normality test, null hypothesis states that data is normally distributed. In R log session below, p-value is greater than 0.05 that means null hypothesis is accepted and hence we can say that the data is normally distributed.

```
1  > shapiro.test(dm$VersionA)
2
3         Shapiro-Wilk normality test
4
5  data:  dm$VersionA
6  W = 0.9619, p-value = 0.8099
7
8  > shapiro.test(dm$VersionB)
9
10        Shapiro-Wilk normality test
11
12 data:  dm$VersionB
13 W = 0.9721, p-value = 0.9312
```

Step 2: Perform student's t-test

```
> t.test(dm$VersionA,dm$VersionB, var.equal=TRUE, paired=FALSE)

        Two Sample t-test

data:  dm$VersionA and dm$VersionB
t = -2.3683, df = 22, p-value = 0.02708
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -15.318196  -1.015137
sample estimates:
mean of x mean of y
 29.83333  38.00000
```

The p-value in the above R log session is below the threshold of 0.05, therefore we conclude that the likelihood of the null hypothesis to be true to be too low. Therefore we must accept the alternate hypothesis i.e. Version A of the web page design is faster as it has a mean completion time that is statistically significantly lower than version B.

T-test can be of two types: two-tailed or one tailed. A two-tailed test is used whenever the result is interesting regardless of direction. For Example: Two versions of a web page are compared for task completion efficiency. A two-tailed test would be used if we don't care if version A is faster than B or vice versa as mentioned in our example scenario.

Type I and Type II errors

Type I Error is an error in judgment when you declare that there's a difference when there really isn't i.e. a false rejection of the null hypothesis. Type II Error is an error in judgment when you declare that there's no difference when there really is i.e. a false acceptance of the null hypothesis.

Non-Parametric tests

When the data is not normally distributed, then a non-parametric equivalent of a test must be used. The Kruskal-Wallis test is a non-parametric equivalent of the t-test. Non-parametric tests are more likely to lead to Type II errors.

Example- Problem - Test if the monthly ozone density in New York has identical data distributions from May to September 1973 using the built-in R dataset named "airquality".

Again the first step remains the same to find out whether the data is normally distributed or not.

```
1 > shapiro.test(airquality$Ozone)
2
3        Shapiro-Wilk normality test
4
5 data:  airquality$Ozone
6 W = 0.8787, p-value = 2.79e-08
```

The p-value in the above R log session is less than 0.05 and hence we fail to accept the null hypothesis of normal distribution and we conclude that this data is not normally distributed.

Based on the the problem statement our null hypothesis will be - "Monthly ozone density in New York has identical data distributions". Now to address the above problem, Kruskal wallis test is performed.

```
1 > kruskal.test(Ozone ~ Month, data = airquality)
2
3        Kruskal-Wallis rank sum test
4
5 data:  Ozone by Month
6 Kruskal-Wallis chi-squared = 29.2666, df = 4, p-value =
7 6.901e-06
```

The p-value in the above R log session is less than 0.05 and hence we fail to accept the null hypothesis of identical data distributions of Ozone density in New York.