Rajanarayanan Thottuvaikkatumana

Apache Spark 2 for Beginners

Develop large-scale distributed data processing applications using Spark 2 in Scala and Python

Packt>

Apache Spark 2 for Beginners

Develop large-scale distributed data processing applications using Spark 2 in Scala and Python

Rajanarayanan Thottuvaikkatumana

Packt>

BIRMINGHAM - MUMBAI

Apache Spark 2 for Beginners

Credits

Author

Rajanarayanan Thottuvaikkatumana

Reviewer

Kornel Skałkowski

Acquisition Editor

Tushar Gupta

Content Development Editor

Samantha Gonsalves

Technical Editor

Jayesh Sonawane

Production Coordinator

Aparna Bhagat

Copy Editor

Safis editing

Project Coordinator

Devanshi Doshi

Proofreader

Safis Editing

Indexer

Rekha Nair

Graphics

Jason Monteiro

About the Author

Rajanarayanan Thottuvaikkatumana, Raj, is a seasoned technologist with more than 23 years of software development experience at various multinational companies. He has lived and worked in India, Singapore, and the USA, and is presently based out of the UK. His experience includes architecting, designing, and developing software applications. He has worked on various technologies including major databases, application development platforms, web technologies, and big data technologies. Since 2000, he has been working mainly in Java related technologies, and does heavy-duty server-side programming in Java and Scala. He has worked on very highly concurrent, highly distributed, and high transaction volume systems. Currently he is building a next generation Hadoop YARN-based data processing platform and an application suite built with Spark using Scala.

Raj holds one master's degree in Mathematics, one master's degree in Computer Information Systems and has many certifications in ITIL and cloud computing to his credit. Raj is the author of *Cassandra Design Patterns - Second Edition*, published by Packt.

When not working on the assignments his day job demands, Raj is an avid listener to classical music and watches a lot of tennis.

About the Reviewer

Kornel Skałkowski has a solid academic and industrial background. For more than five years, he worked as an assistant at AGH University of Science and Technology in Krakow. In 2015, he obtained his Ph.D. in the subject of machine learning-based adaptation of SOA systems. He has cooperated with several companies on various projects concerning intelligent systems, machine learning and big data. Currently, he works as a big data developer for SAP SE.

He is a co-author of 19 papers concerning software engineering, SOA systems and machine learning. He also works as a reviewer for the American Journal of Software Engineering and Applications. He has participated in numerous European and national scientific projects. His research interests include machine learning, big data and software engineering.

He is author of the book *Data Lake Development for Big Data*.

> *I would like to kindly thank my family, my relatives and my friends for their endless patience and support during reviewing this book. I would also like to express my special gratitude to my girlfriend Ania, for her understanding us missing time together.*

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Dedicating this book to the countless volunteers who worked tirelessly to build high production-quality open source software products. Without them I wouldn't have written this book.

Table of Contents

Preface

The data processing framework named Spark was first built to prove that, by re-using the data sets across a number of iterations, it provided value where Hadoop MapReduce jobs performed poorly. The research paper *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center* talks about the philosophy behind the design of Spark. A very simplistic reference implementation built to test Mesos by the University of California Berkeley researchers has grown far and beyond to become a full blown data processing framework later became one of the most active Apache projects. It is designed from the ground up to do distributed data processing on clusters such as Hadoop, Mesos, and in standalone mode. Spark is a JVM-based data processing framework and hence it works on most operating systems that support JVM-based applications. Spark is widely installed on UNIX and Mac OS X, platforms and Windows adoption is increasing.

Spark provides a unified programming model using the programming languages Scala, Java, Python and R. In other words, irrespective of the language used to program Spark applications, the API remains almost the same in all the languages. In this way, organizations can adopt Spark and develop applications in their programming language of choice. This also enables fast porting of Spark applications from one language to another without much effort, if there is a need. Most of Spark is developed using Scala and because of that the Spark programming model inherently supports functional programming principles. The most basic Spark data abstraction is the resilient distributed data set (RDD), based on which all the other libraries are built. The RDD-based Spark programming model is the lowest level where developers can build data processing applications.

Spark has grown fast, to cater to the needs of more data processing use cases. When such a forward-looking step is taken with respect to the product road map, the requirement emerged to make the programming more high level for business users. The Spark SQL library on top of Spark Core, with its DataFrame abstraction, was built to cater to the needs of the huge population of developers who are very conversant with the ubiquitous SQL.

Data scientists use R for their computation needs. The biggest limitation of R is that all the data that needs to be processed should *fit* into the main memory of the computer on which the R program is running. The R API for Spark introduced data scientists to the world of distributed data processing in their familiar data frame abstraction. In other words, using the R API for Spark, the processing of data can be done in parallel on Hadoop or Mesos, growing far beyond the limitation of the resident memory of the host computer.

In the present era of large-scale applications that collect data, the velocity of the data that is ingested is very high. Many application use cases mandate real-time processing of the data that is streamed. The Spark Streaming library, built on top of Spark Core, does exactly the same.

The data at rest or the data that is streamed are fed to machine learning algorithms to train data models and use them to provide answers to business questions. All the machine learning frameworks that were created before Spark had many limitations in terms of the memory of the processing computer, inability to do parallel processing, repeated read-write cycles, so on. Spark doesn't have any of these limitations and hence the Spark MLlib machine learning library, built on top of Spark Core and Spark DataFrames, turned out to be the best of breed machine learning library that glues together the data processing pipelines and machine learning activities.

Graph is a very useful data structure used heavily in some special use cases. The algorithms used to process the data in a graph data structure are computationally intensive. Before Spark, many graph processing frameworks came along, and some of them were really fast at processing, but pre-processing the data needed to produce the graph data structure turned out to be a big bottleneck in most of these graph processing applications. The Spark GraphX library, built on top of Spark, filled this gap to make data processing and graph processing as chained activities.

In the past, many data processing frameworks existed and many of them were proprietary forcing organizations to get into the trap of vendor lock-in. Spark provided a very viable alternative for a wide variety of data processing needs with no licensing cost; at the same time, it was backed by many leading companies, providing professional production support.

What this book covers

`Chapter 1`, *Spark Fundamentals*, discusses the fundamentals of Spark as a framework with its APIs and the libraries that comes with it, along with the whole data processing ecosystem Spark is interacting with.

`Chapter 2`, *Spark Programming Model*, discusses the uniform programming model, based on the tenets of functional programming methodology, that is used in Spark, and covers the fundamentals of resilient distributed data sets (RDD), Spark transformations, and Spark actions.

`Chapter 3`, *Spark SQL*, discusses Spark SQL, which is one of the most powerful Spark libraries used to manipulate data using the ubiquitous SQL constructs in conjunction with the Spark DataFrame API, and and how it works with Spark programs. This chapter also discusses how Spark SQL is used to access data from various data sources, enabling the unification of diverse data sources for data processing.

`Chapter 4`, *Spark Programming with R*, discusses SparkR or R on Spark, which is the R API for Spark; this enables R users to make use of the data processing capabilities of Spark using their familiar data frame abstraction. It gives a very good foundation for R users to get acquainted with the Spark data processing ecosystem.

`Chapter 5`, *Spark Data Analysis with Python*, discusses the use of Spark to do data processing and Python to do data analysis, using a wide variety of charting and plotting libraries available for Python. This chapter discusses combining these two related activities together as a Spark application with Python as the programming language of choice.

`Chapter 6`, *Spark Stream Processing*, discusses Spark Streaming, which is one of the most powerful Spark libraries to capture and process data that is ingested as a stream. Kafka as the distributed message broker and a Spark Streaming application as the consumer are also discussed.

`Chapter 7`, *Spark Machine Learning*, discusses Spark MLlib, which is one of the most powerful Spark libraries used to develop machine learning applications at an introductory level.

`Chapter 8`, *Spark Graph Processing*, discusses Spark GraphX, which is one of the most powerful Spark libraries to process graph data structures, and comes with lots of algorithms to process data in graphs. This chapter covers the basics of GraphX and some use cases implemented using the algorithms provided by GraphX.

`Chapter 9`, *Designing Spark Applications*, discusses the design and development of a Spark data processing application, covering various features of Spark that were covered in the previous chapters of this book.

What you need for this book

Spark 2.0.0 or above is to be installed on at least a standalone machine to run the code samples and do further activities to learn more about the subject. For `Chapter 6`, Spark Stream Processing, Kafka needs to be installed and configured as a message broker with its command line producer producing messages and the application developed using Spark as a consumer of those messages.

Who this book is for

If you are an application developer, data scientist, or big data solutions architect who is interested in combining the data processing power of Spark with R, and consolidating data processing, stream processing, machine learning, and graph processing into one unified and highly interoperable framework with a uniform API using Scala or Python, this book is for you.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: " It is a good idea to customize this property `spark.driver.memory` to have a higher value."

A block of code is set as follows:

```
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
```

Any command-line input or output is written as follows:

```
$ python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The shortcuts in this book are based on the `Mac OS X 10.5+` scheme."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Apache-Spark-2-for-Beginners`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/ApacheSpark2forBeginners_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Spark Fundamentals

Data is one of the most important assets of any organization. The scale at which data is being collected and used in organizations is growing beyond imagination. The speed at which data is being ingested, the variety of the data types in use, and the amount of data that is being processed and stored are breaking all-time records every moment. It is very common these days, even in small-scale organizations, that data is growing from gigabytes to terabytes to petabytes. For the same reason, the processing needs are also growing that ask for capability to process data at rest as well as data on the move.

Take any organization; its success depends on the decisions made by its leaders and for making sound decisions, you need the backing of good data and the information generated by processing the data. This poses a big challenge on how to process the data in a timely and cost-effective manner so that right decisions can be made. Data processing techniques have evolved since the early days of computers. Countless data processing products and frameworks came into the market and disappeared over these years. Most of these data processing products and frameworks were not general purpose in nature. Most of the organizations relied on their own bespoke applications for their data processing needs, in a silo way, or in conjunction with specific products.

Large-scale Internet applications, popularly known as **Internet of Things** (**IoT**) applications, heralded the common need to have open frameworks to process huge amounts of data ingested at great speed dealing with various types of data. Large-scale web sites, media streaming applications, and the huge batch processing needs of organizations made the need even more relevant. The open source community is also growing considerably along with the growth of the Internet, delivering production quality software supported by reputed software companies. A huge number of companies started using open source software and started deploying them in their production environments.

In a technological perspective, the data processing needs were facing huge challenges. The amount of data started overflowing from single machines to clusters of huge numbers of machines. The processing power of the single CPU plateaued and modern computers started combining them together to get more processing power, known as multi-core computers. The applications were not designed and developed to make use of all the processors in a multi-core computer and wasted lots of the processing power available in a typical modern computer.

> Throughout this book, the terms *node*, *host*, and *machine* refer to a computer that is running in a standalone mode or in a cluster.

In this context, what are the qualities an ideal data processing framework should possess?

- It should be capable of processing the blocks of data distributed across a cluster of computers
- It should be able to process the data in a parallel fashion so that a huge data processing job can be divided into multiple tasks processed in parallel so that the processing time can be reduced considerably
- It should be capable of using the processing power of all the cores or processors in a computer
- It should be capable of using all the available computers in a cluster
- It should be capable of running on commodity hardware

There are two open source data processing frameworks that are worth mentioning that satisfy all these requirements. The first is being Apache Hadoop and the second one is Apache Spark.

We will cover the following topics in this chapter:

- Apache Hadoop
- Apache Spark
- Spark 2.0 installation

An overview of Apache Hadoop

Apache Hadoop is an open source software framework designed from ground-up to do distributed data storage on a cluster of computers and to do distributed data processing of the data that is spread across the cluster of computers. This framework comes with a distributed filesystem for the data storage, namely, **Hadoop Distributed File System** (**HDFS**), and a data processing framework, namely, MapReduce. The creation of HDFS is inspired from the Google research paper, *The Google File System* and MapReduce is based on the Google research paper, *MapReduce: Simplified Data Processing on Large Clusters*.

Hadoop was adopted by organizations in a really big way by implementing huge Hadoop clusters for data processing. It saw tremendous growth from Hadoop MapReduce version 1 (MRv1) to Hadoop MapReduce version 2 (MRv2). From a pure data processing perspective, MRv1 consisted of HDFS and MapReduce as the core components. Many applications, generally called SQL-on-Hadoop applications, such as Hive and Pig, were stacked on top of the MapReduce framework. It is very common to see that even though these types of applications are separate Apache projects, as a suite, many such projects provide great value.

The **Yet Another Resource Negotiator** (**YARN**) project came to the fore with computing frameworks other than MapReduce type to run on the Hadoop ecosystem. With the introduction of YARN sitting on top of HDFS, and below MapReduce in a component architecture layering perspective, the users could write their own applications that can run on YARN and HDFS to make use of the distributed data storage and data processing capabilities of the Hadoop ecosystem. In other words, the newly overhauled MapReduce version 2 (MRv2) became one of the application frameworks sitting on top of HDFS and YARN.

Figure 1 gives a brief idea about these components and how they are stacked together:
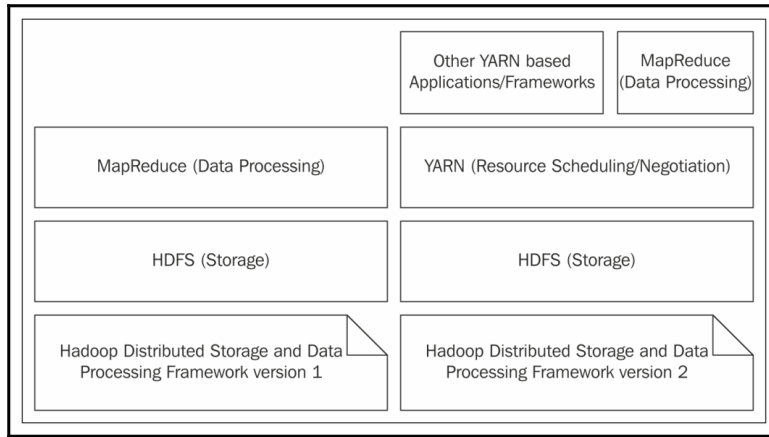
Figure 1

MapReduce is a generic data processing model. The data processing goes through two steps, namely, *map* step and *reduce* step. In the first step, the input data is divided into a number of smaller parts so that each one of them can be processed independently. Once the *map* step is completed, its output is consolidated and the final result is generated in the *reduce* step. In a typical word count example, the creation of key-value pairs with each word as the key and the value 1 is the *map* step. The sorting of these pairs on the key, summing the values of the pairs with the same key falls into an intermediate *combine* step. Producing the pairs containing unique words and their occurrence count is the *reduce* step.

From an application programming perspective, the basic ingredients for an over-simplified MapReduce application are as follows:

- Input location
- Output location
- Map function implemented for the data processing need from the appropriate interfaces and classes from the `MapReduce` library
- Reduce function implemented for the data processing need from the appropriate interfaces and classes from the `MapReduce` library

The MapReduce job is submitted for running in Hadoop and once the job is completed, the output can be taken from the output location specified.

This two-step process of dividing a `MapReduce` data processing job to *map* and *reduce* tasks was highly effective and turned out to be a perfect fit for many batch data processing use cases. There is a lot of Input/Output (I/O) operations with the disk happening under the hood during the whole process. Even in the intermediate steps of the MapReduce job, if the internal data structures are filled with data or when the tasks are completed beyond a certain percentage, writing to the disk happens. Because of this, the subsequent steps in the MapReduce jobs have to read from the disk.

Then the other biggest challenge comes when there are multiple MapReduce jobs to be completed in a chained fashion. In other words, if a big data processing work is to be accomplished by two MapReduce jobs in such a way that the output of the first MapReduce job is the input of the second MapReduce job. In this situation, whatever may be the size of the output of the first MapReduce job, it has to be written to the disk before the second MapReduce could use it as its input. So in this simple case, there is a definite and *unnecessary* write operation.

In many of the batch data processing use cases, these I/O operations are not a big issue. If the results are highly reliable, for many batch data processing use cases, latency is tolerated. But the biggest challenge comes when doing real-time data processing. The huge amount of I/O operations involved in MapReduce jobs makes it unsuitable for real-time data processing with the lowest possible latency.

Understanding Apache Spark

Spark is a **Java Virtual Machine** (**JVM**) based distributed data processing engine that scales, and it is fast compared to many other data processing frameworks. Spark was originated at the *University of California Berkeley* and later became one of the top projects in Apache. The research paper, *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*, talks about the philosophy behind the design of Spark. The research paper states:

> *"To test the hypothesis that simple specialized frameworks provide value, we identified one class of jobs that were found to perform poorly on Hadoop by machine learning researchers at our lab: iterative jobs, where a dataset is reused across a number of iterations. We built a specialized framework called Spark optimized for these workloads."*

The biggest claim from Spark regarding speed is that it is able to *"Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk".* Spark could make this claim because it does the processing in the main memory of the worker nodes and prevents the *unnecessary* I/O operations with the disks. The other advantage Spark offers is the ability to chain the tasks even at an application programming level without writing onto the disks at all or minimizing the number of writes to the disks.

How did Spark become so efficient in data processing as compared to MapReduce? It comes with a very advanced **Directed Acyclic Graph** (**DAG**) data processing engine. What it means is that for every Spark job, a DAG of tasks is created to be executed by the engine. The DAG in mathematical parlance consists of a set of vertices and directed edges connecting them. The tasks are executed as per the DAG layout. In the MapReduce case, the DAG consists of only two vertices, with one vertex for the *map* task and the other one for the *reduce* task. The edge is directed from the *map* vertex to the *reduce* vertex. The in-memory data processing combined with its DAG-based data processing engine makes Spark very efficient. In Spark's case, the DAG of tasks can be as complicated as it can. Thankfully, Spark comes with utilities that can give excellent visualization of the DAG of any Spark job that is running. In a word count example, Spark's Scala code will look something like the following code snippet . The details of this programming aspects will be covered in the coming chapters:

```
val textFile = sc.textFile("README.md")
val wordCounts = textFile.flatMap(line => line.split(" ")).map(word =>
 (word, 1)).reduceByKey((a, b) => a + b)
wordCounts.collect()
```

The web application that comes with Spark is capable of monitoring the workers and applications. The DAG of the preceding Spark job generated on the fly will look like *Figure 2*, as shown here:
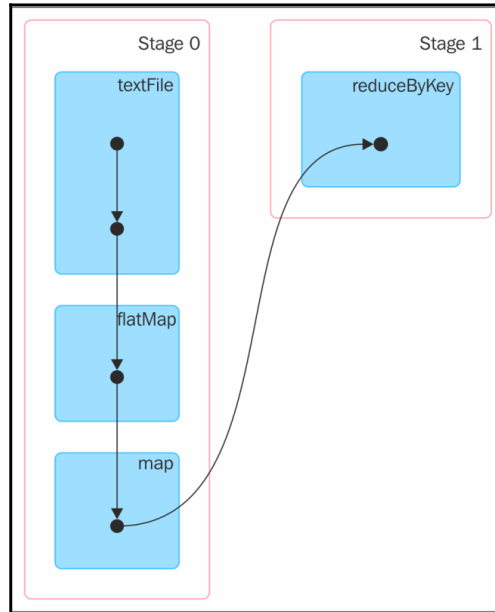
Figure 2

The Spark programming paradigm is very powerful and exposes a uniform programming model supporting the application development in multiple programming languages. Spark supports programming in Scala, Java, Python, and R even though there is no functional parity across all the programming languages supported. Apart from writing Spark applications in these programming languages, Spark has an interactive shell with **Read, Evaluate, Print, and Loop** (**REPL**) capabilities for the programming languages Scala, Python, and R. At this moment, there is no REPL support for Java in Spark. The Spark REPL is a very versatile tool that can be used to try and test Spark application code in an interactive fashion. The Spark REPL enables easy prototyping, debugging, and much more.

In addition to the core data processing engine, Spark comes with a powerful stack of domain specific libraries that use the core Spark libraries and provide various functionalities useful for various big data processing needs. The following table lists the supported libraries:

Library	Use	Supported Languages
Spark SQL	Enables the use of SQL statements or DataFrame API inside Spark applications	Scala, Java, Python, and R
Spark Streaming	Enables processing of live data streams	Scala, Java, and Python
Spark MLlib	Enables development of machine learning applications	Scala, Java, Python, and R
Spark GraphX	Enables graph processing and supports a growing library of graph algorithms	Scala

Spark can be deployed on a variety of platforms. Spark runs on the **operating systems** (**OS**) Windows and UNIX (such as Linux and Mac OS). Spark can be deployed in a standalone mode on a single node having a supported OS. Spark can also be deployed in cluster node on Hadoop YARN as well as Apache Mesos. Spark can be deployed in the Amazon EC2 cloud as well. Spark can access data from a wide variety of data stores, and some of the most popular ones include HDFS, Apache Cassandra, Hbase, Hive, and so on. Apart from the previously listed data stores, if there is a driver or connector program available, Spark can access data from pretty much any data source.

> All the examples used in this book are developed, tested, and run on a Mac OS X Version 10.9.5 computer. The same instructions are applicable for all the other platforms except Windows. In Windows, corresponding to all the UNIX commands, there is a file with a `.cmd` extension and it has to be used. For example, for `spark-shell` in UNIX, there is a `spark-shell.cmd` in Windows. The program behavior and results should be the same across all the supported OS.

In any distributed application, it is common to have a driver program that controls the execution and there will be one or more worker nodes. The driver program allocates the tasks to the appropriate workers. This is the same even if Spark is running in standalone mode. In the case of a Spark application, its **SparkContext** object is the driver program and it communicates with the appropriate cluster manager to run the tasks. The Spark master, which is part of the Spark core library, the Mesos master, and the Hadoop YARN Resource Manager, are some of the cluster managers that Spark supports. In the case of a Hadoop YARN deployment of Spark, the Spark driver program runs inside the Hadoop YARN application master process or the Spark driver program runs as a client to the Hadoop YARN. *Figure 3* describes the standalone deployment of Spark:
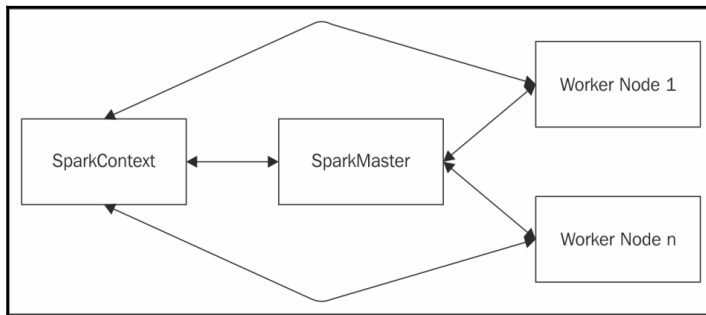
Figure 3

In the Mesos deployment mode of Spark, the cluster manager will be the**Mesos Master**. *Figure 4* describes the Mesos deployment of Spark:
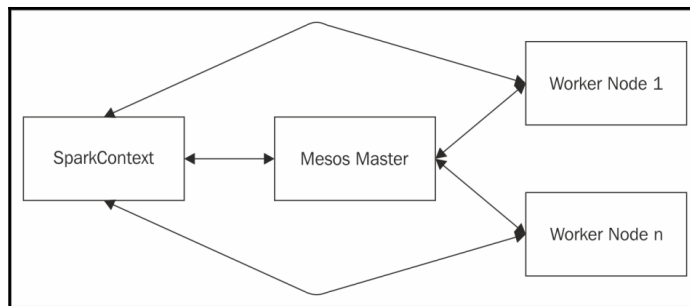
Figure 4

In the Hadoop YARN deployment mode of Spark, the cluster manager will be the Hadoop Resource Manager and its address will be picked up from the Hadoop configuration. In other words, when submitting the Spark jobs, there is no need to give an explicit master URL and it will pick up the details of the cluster manager from the Hadoop configuration. *Figure 5* describes the Hadoop YARN deployment of Spark:
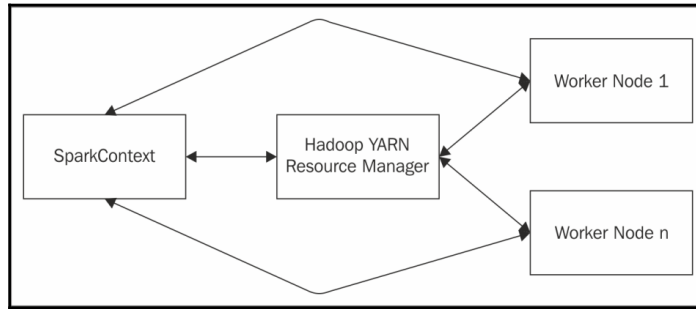
Figure 5

Spark runs in the cloud too. In the case of the deployment of Spark on Amazon EC2, apart from accessing the data from the regular supported data sources, Spark can also access data from Amazon S3, which is the online data storage service from Amazon.

Installing Spark on your machines

Spark supports application development in Scala, Java, Python, and R. In this book, Scala, Python, and R, are used. Here is the reason behind the choice of the languages for the examples in this book. The Spark interactive shell, or REPL, allows the user to execute programs on the fly just like entering OS commands on a terminal prompt and it is available only for the languages Scala, Python and R. REPL is the best way to try out Spark code before putting them together in a file and running them as applications. REPL helps even the experienced programmer to try and test the code and thus facilitates fast prototyping. So, especially for beginners, using REPL is the best way to get started with Spark.

As a pre-requisite to Spark installation and to do Spark programming in Python and R, both Python and R are to be installed prior to the installation of Spark.

Python installation

Visit `https://www.python.org` for downloading and installing Python for your computer. Once the installation is complete, make sure that the required binaries are in the OS search path and the Python interactive shell is coming up properly. The shell should display some content similar to the following:

```
$ python
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

For charting and plotting, the `matplotlib` library is being used.

> Python version 3.5.0 is used as a version of choice for Python. Even though Spark supports programming in Python version 2.7, as a forward looking practice, the latest and most stable version of Python available is used. Moreover, most of the important libraries are getting ported to Python version 3.x as well.

Visit `http://matplotlib.org` for downloading and installing the library. To make sure that the library is installed properly and that charts and plots are getting displayed properly, visit the `http://matplotlib.org/examples/index.html` page to pick up some example code and see that your computer has all the required resources and components for charting and plotting. While trying to run some of these charting and plotting samples, in the context of the import of the libraries in Python code, there is a possibility that it may complain about the missing locale. In that case, set the following environment variables in the appropriate user profile to get rid of the error messages:

```
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8
```

R installation

Visit `https://www.r-project.org` for downloading and installing R for your computer. Once the installation is complete, make sure that the required binaries are in the OS search path and the R interactive shell is coming up properly. The shell should display some content similar to the following:

```
$ r
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
```

[18]

```
Platform: x86_64-apple-darwin13.4.0 (64-bit)
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
  Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Previously saved workspace restored]
>
```

R version 3.2.2 is the choice for R.

Spark installation

Spark installation can be done in many different ways. The most important pre-requisite for Spark installation is that the Java 1.8 JDK is installed in the system and the JAVA_HOME environment variable is set to point to the Java 1.8 JDK installation directory. Visit http://spark.apache.org/downloads.html for understanding, choosing, and downloading the right type of installation for your computer. Spark version 2.0.0 is the version of choice for following the examples given in this book. Anyone who is interested in building and using Spark from the source code should visit: http://spark.apache.org/docs/latest/building-spark.html for the instructions. By default, when you build Spark from the source code, it will not build the R libraries for Spark. For that, the SparkR libraries have to be built and the appropriate profile has to be included while building Spark from source code. The following command shows how to include the profile required to build the SparkR libraries:

```
$ mvn -DskipTests -Psparkr clean package
```

Once the Spark installation is complete, define the following environment variables in the appropriate user profile:

```
export SPARK_HOME=<the Spark installation directory>
export PATH=$SPARK_HOME/bin:$PATH
```

If there are multiple versions of Python executables in the system, then it is better to explicitly specify the Python executable to be used by Spark in the following environment variable setting:

```
export PYSPARK_PYTHON=/usr/bin/python
```

In the `$SPARK_HOME/bin/pyspark` script, there is a block of code that determines the Python executable to be used by Spark:

```
# Determine the Python executable to use if PYSPARK_PYTHON or
PYSPARK_DRIVER_PYTHON isn't set:
if hash python2.7 2>/dev/null; then
  # Attempt to use Python 2.7, if installed:
  DEFAULT_PYTHON="python2.7"
else
  DEFAULT_PYTHON="python"
fi
```

So, it is always better to explicitly set the Python executable for Spark, even if there is only one version of Python available in the system. This is a safeguard to prevent unexpected behavior when an additional version of Python is installed in the future.

Once all the preceding steps are completed successfully, make sure that all the Spark shells for the languages Scala, Python, and R are working properly. Run the following commands on the OS terminal prompt and make sure that there are no errors and that content similar to the following is getting displayed. The following set of commands is used to bring up the Scala REPL of Spark:

```
$ cd $SPARK_HOME
$ ./bin/spark-shellUsing Spark's default log4j profile:
org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
16/06/28 20:53:48 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
16/06/28 20:53:49 WARN SparkContext: Use an existing SparkContext, some
configuration may not take effect.
Spark context Web UI available at http://192.168.1.6:4040
Spark context available as 'sc' (master = local[*], app id =
local-1467143629623).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.0.1
      /_/
```

```
Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_66)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
scala>exit
```

In the preceding display, verify that the JDK version, Scala version, and Spark version are correct as per the settings in the computer in which Spark is installed. The most important point to verify is that no error messages are displayed.

The following set of commands is used to bring up the Python REPL of Spark:

```
$ cd $SPARK_HOME
$ ./bin/pyspark
Python 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015, 11:00:19)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
16/06/28 20:58:04 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.0.1
      /_/

Using Python version 3.5.0 (v3.5.0:374f501f4567, Sep 12 2015 11:00:19)
SparkSession available as 'spark'.
>>>exit()
```
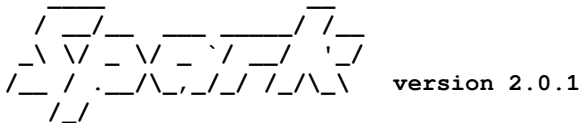
In the preceding display, verify that the Python version, and Spark version are correct as per the settings in the computer in which Spark is installed. The most important point to verify is that no error messages are displayed.

The following set of commands are used to bring up the R REPL of Spark:

```
$ cd $SPARK_HOME
$ ./bin/sparkR
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)
```

```
    R is free software and comes with ABSOLUTELY NO WARRANTY.
    You are welcome to redistribute it under certain conditions.
    Type 'license()' or 'licence()' for distribution details.

      Natural language support but running in an English locale

    R is a collaborative project with many contributors.
    Type 'contributors()' for more information and
    'citation()' on how to cite R or R packages in publications.

    Type 'demo()' for some demos, 'help()' for on-line help, or
    'help.start()' for an HTML browser interface to help.
    Type 'q()' to quit R.

    [Previously saved workspace restored]

    Launching java with spark-submit command /Users/RajT/source-code/spark-
    source/spark-2.0/bin/spark-submit    "sparkr-shell"
    /var/folders/nf/trtmyt9534z03kq8p8zgbnxh0000gn/T//RtmphPJkkF/backend_port59
    418b49bb6
    Using Spark's default log4j profile: org/apache/spark/log4j-
    defaults.properties
    Setting default log level to "WARN".
    To adjust logging level use sc.setLogLevel(newLevel).
    16/06/28 21:00:35 WARN NativeCodeLoader: Unable to load native-hadoop
    library for your platform... using builtin-java classes where applicable

     Welcome to
          ____              __
         / __/__  ___ _____/ /__
        _\ \/ _ \/ _ `/ __/  '_/
       /___/ .__/\_,_/_/ /_/\_\   version  2.0.1
          /_/


     Spark context is available as sc, SQL context is available as sqlContext
    During startup - Warning messages:
    1: 'SparkR::sparkR.init' is deprecated.
    Use 'sparkR.session' instead.
    See help("Deprecated")
    2: 'SparkR::sparkRSQL.init' is deprecated.
    Use 'sparkR.session' instead.
    See help("Deprecated")
    >q()
```

In the preceding display, verify that the R version and Spark version are correct as per the settings in the computer in which Spark is installed. The most important point to verify is that no error messages are displayed.

If all the REPL for Scala, Python, and R are working fine, it is almost certain that the Spark installation is good. As a final test, run some of the example programs that came with Spark and make sure that they are giving proper results close to the results shown below the commands and not throwing any error messages in the console. When these example programs are run, apart from the output shown below the commands, there will be lot of other messages displayed in the console. They are omitted to focus on the results:

```
$ cd $SPARK_HOME
$ ./bin/run-example SparkPi
Pi is roughly 3.1484
$ ./bin/spark-submit examples/src/main/python/pi.py
Pi is roughly 3.138680
$ ./bin/spark-submit examples/src/main/r/dataframe.R
root
 |-- name: string (nullable = true)
 |-- age: double (nullable = true)
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
    name
1 Justin
```

Development tool installation

Most of the code that is going to be discussed in this book can be tried and tested in the appropriate REPL. But the proper Spark application development is not possible without some basic build tools. As a bare minimum requirement, for developing and building Spark applications in Scala, the **Scala build tool** (**sbt**) is a must. Visit http://www.scala-sbt.org for downloading and installing sbt.

Maven is the preferred build tool for building Java applications. This book is not talking about Spark application development in Java, but it is good to have Maven also installed in the system. Maven will come in handy if Spark is to be built from source. Visit https://maven.apache.org for downloading and installing Maven.

There are many **Integrated Development Environments** (**IDEs**) available for Scala as well as Java. It is a personal choice, and the developer can choose the tool of his/her choice for the language in which he/she is developing Spark applications.

Optional software installation

Spark REPL for Scala is a good start to get into the prototyping and testing of some small snippets of code. But when there is a need to develop, build, and package Spark applications in Scala, it is good to have sbt-based Scala projects and develop them using a supported IDE, including but not limited to Eclipse or IntelliJ IDEA. Visit the appropriate website for downloading and installing the preferred IDE for Scala.

Notebook style application development tools are very common these days among data analysts and researchers. This is akin to a lab notebook. In a typical lab notebook, there will be instructions, detailed descriptions, and steps to follow to conduct an experiment. Then the experiments are conducted. Once the experiments are completed, there will be results captured in the notebook. If all these constructs are combined together and fit into the context of a software program and modeled in a lab notebook format, there will be documentation, code, input, and the output generated by running the code. This will give a very good effect, especially if the programs generate a lot of charts and plots.

> For those who are not familiar with notebook style application development IDEs, there is a very nice article entitled *Interactive Notebooks: Sharing the Code* that can be read from `http://www.nature.com/news/interactive-notebooks-sharing-the-code-1.16261`. As an optional software development IDE for Python, the IPython notebook is described in the following section. After the installation, get yourself familiar with the tool before getting into serious development with it.

IPython

In the case of Spark application development in Python, IPython provides an excellent notebook-style development tool, which is a Python language kernel for Jupyter. Spark can be integrated with IPython, so that when the Spark REPL for Python is invoked, it will start the IPython notebook. Then, create a notebook and start writing code in the notebook just like the way commands are given in the Spark REPL for Python. Visit `http://ipython.org` to download and install the IPython notebook. Once the installation is complete, invoke the IPython notebook interface and make sure that some example Python code is running fine. Invoke commands from the directory from where the notebooks are stored or where the notebooks are to be stored. Here, the IPython notebook is started from a temporary directory. When the following commands are invoked, it will open up the web interface and from there create a new notebook by clicking the New drop-down box and picking up the appropriate Python version.

The following screenshot shows how to combine a markdown style documentation, a Python program, and the generated output together in an IPython notebook:

```
$ cd /Users/RajT/temp
$ ipython notebook
```
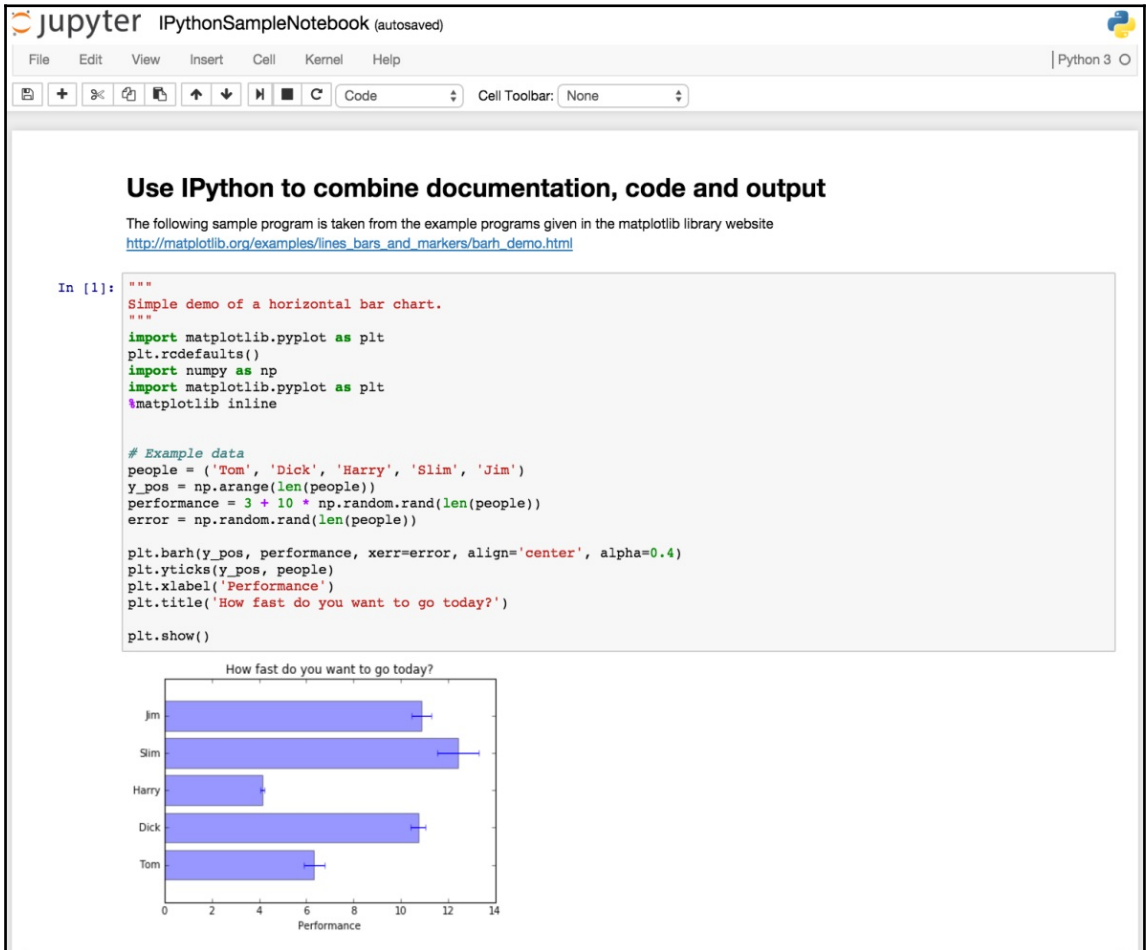

Figure 6

Figure 6 shows how the IPython notebook can be used to write simple Python programs. The IPython notebook can be configured as a shell of choice for Spark, and when the Spark REPL for Python is invoked, it will start up the IPython notebook and Spark application development can be done using IPython notebook. To achieve that, define the following environment variables in the appropriate user profile:

```
export PYSPARK_DRIVER_PYTHON=ipython
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
```

Now, instead of invoking the IPython notebook from the command prompt, invoke the Spark REPL for Python. Just like what has been done before, create a new IPython notebook and start writing Spark code in Python:

```
$ cd /Users/RajT/temp
$ pyspark
```

Take a look at the following screenshot:
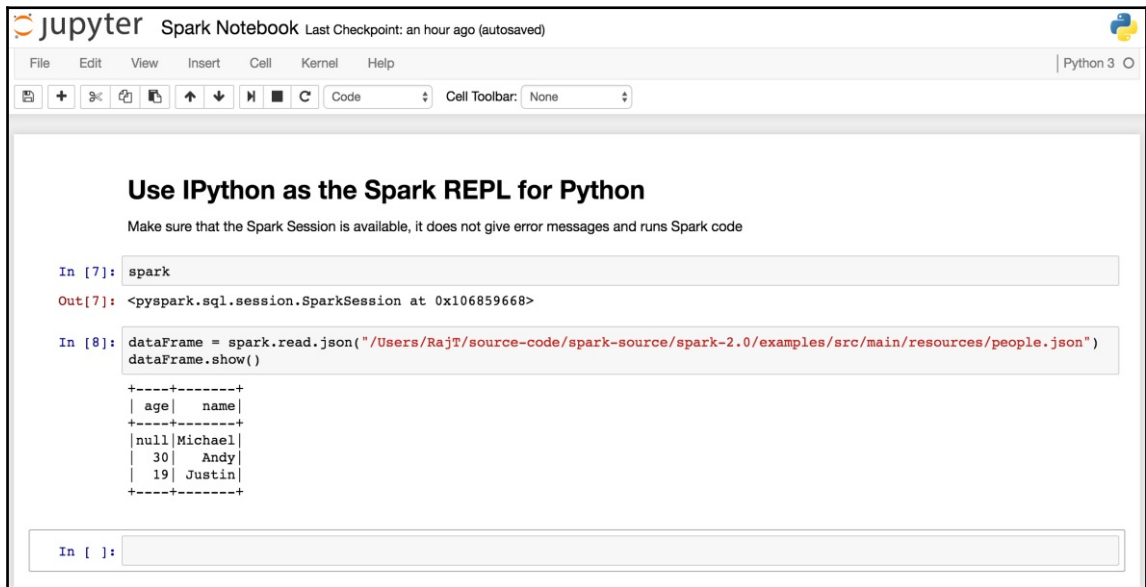
Figure 7

In the standard Spark REPL for any language, it is possible to refer the files located in the local filesystem with their relative path. When the IPython notebook is being used, local files are to be referred with their full path.

RStudio

Among the R user community, the preferred IDE for R is the RStudio. RStudio can be used to develop Spark applications in R as well. Visit `https://www.rstudio.com` to download and install RStudio. Once the installation is complete, before running any Spark R code, it is mandatory to include the `SparkR` libraries and set some variables to make sure that the Spark R programs are running smoothly from RStudio. The following code snippet does that:

```
SPARK_HOME_DIR <- "/Users/RajT/source-code/spark-source/spark-2.0"
Sys.setenv(SPARK_HOME=SPARK_HOME_DIR)
.libPaths(c(file.path(Sys.getenv("SPARK_HOME"), "R", "lib"), .libPaths()))
library(SparkR)
spark <- sparkR.session(master="local[*]")
```

In the preceding R code, change the `SPARK_HOME_DIR` variable definition to point to the directory where Spark is installed. *Figure 8* shows a sample run of the Spark R code from RStudio:
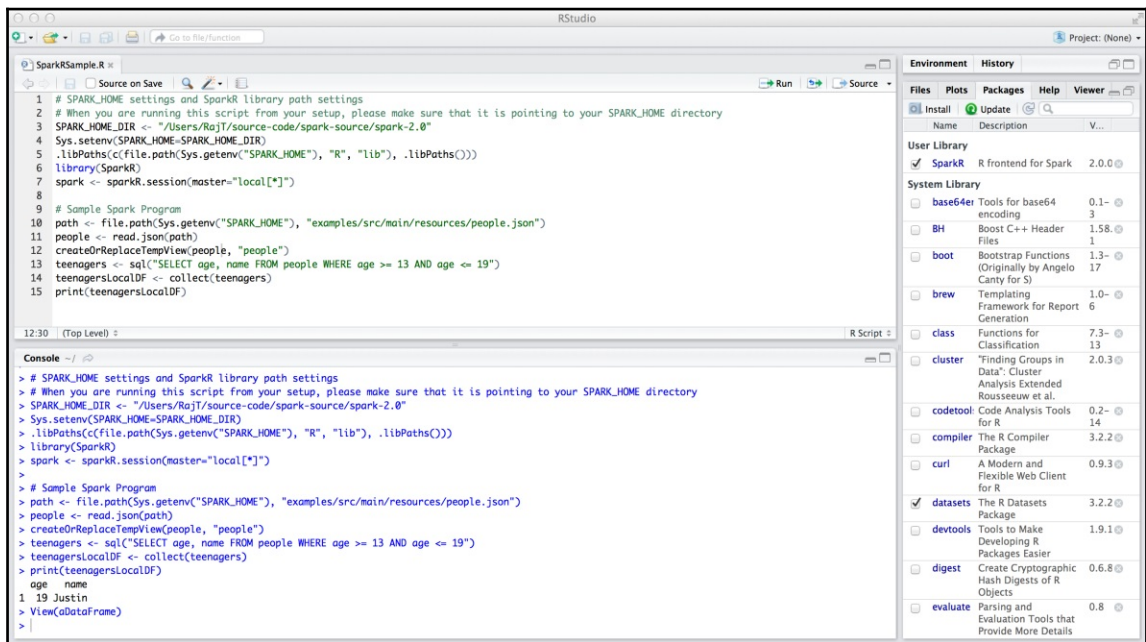
Figure 8

Once all the required software is installed, configured, and working as per the details given previously, the stage is set for Spark application development in Scala, Python, and R.

> The Jupyter notebook supports multiple languages through the custom kernel implementation strategy for various languages. There is a native R kernel, namely IRkernel, for Jupyter which can be installed as an R package.

Apache Zeppelin

Apache Zeppelin is another promising project that is getting incubated right now. It is a web-based notebook similar to Jupyter but supporting multiple languages, shells, and technologies through its interpreter strategy enabling Spark application development inherently. Right now it is in its infant stage, but it has a lot of potential to become one of the best notebook-based application development platforms. Zeppelin has very powerful built-in charting and plotting capabilities using the data generated by the programs written in the notebook.

Zeppelin is built with high extensibility having the ability to plug in many types of interpreters using its Interpreter Framework. End users, just like any other notebook-based system, enter various commands in the notebook interface. These commands are to be processed by some interpreter to generate the output. Unlike many other notebook-style systems, Zeppelin supports a good number of interpreters or backends out of the box such as Spark, Spark SQL, Shell, Markdown, and many more. In terms of the frontend, again it is a pluggable architecture, namely, the **Helium Framework**. The data generated by the backend is displayed by the frontend components such as Angular JS. There are various options to display the data in tabular format, raw format as generated by the interpreters, charts, and plots. Because of the architectural separation of concerns such as the backend, the frontend, and the ability to plug in various components, it is a great way to choose heterogeneous components for the right job. At the same time, it integrates very well to provide a harmonious end-user-friendly data processing ecosystem. Even though there is pluggable architecture capability for various components in Zeppelin, the visualizations are limited. In other words, there are only a few charting and plotting options available out of the box in Zeppelin. Once the notebooks are working fine and producing the expected results, typically, the notebooks are shared with other people and for that, the notebooks are to be persisted. Zeppelin is different again here and it has a highly versatile notebook storage system. The notebooks can be persisted to the filesystem, Amazon S3, or Git, and other storage targets can be added if required.

Platform as a Service (**PaaS**) has been evolving over the last couple of years since the massive innovations happening around Cloud as an application development and deployment platform. For software developers, there are many PaaS platforms available delivered through Cloud, which obviates the need for them to have their own application development stack. Databricks has introduced a Cloud-based big data platform in which users can have access to a notebook-based Spark application development interface in conjunction with micro-cluster infrastructure to which the Spark applications can be submitted. There is a community edition as well, catering to the needs of a wider development community. The biggest advantage of this PaaS platform is that it is a browser-based interface and users can run their code against multiple versions of Spark and on different types of clusters.

References

For more information please refer to the following links:

- `http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp23.pdf`
- `http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi4.pdf`
- `https://www.cs.berkeley.edu/~alig/papers/mesos.pdf`
- `http://spark.apache.org/`
- `https://jupyter.org/`
- `https://github.com/IRkernel/IRkernel`
- `https://zeppelin.incubator.apache.org/`
- `https://community.cloud.databricks.com/`

Summary

Spark is a very powerful data processing platform supporting a uniform programming model. It supports application development in Scala, Java, Python, and R, providing a stack of highly interoperable libraries used for various types of data processing needs, and a plethora of third-party libraries that make use of the Spark ecosystem covering various other data processing use cases. This chapter gave a brief introduction to Spark and setting up the development environment for the Spark application development that is going to be covered in forthcoming chapters of the book.

The next chapter is going to discuss the Spark programming model, the basic abstractions and terminologies, Spark transformations, and Spark actions, in conjunction with real-world use cases.

2
Spark Programming Model

Extract, **Transform**, and **Load** (**ETL**) tools proliferated along with the growth of the data in organizations. The need to move data from one source to one or more destinations, processing it on the fly before it reaches its destination, were all the requirements of the time. Most of the time, these ETL tools were supporting only a few types of data, only a few types of data sources and destinations, and were closed to extension to allow them to support new data types and new sources and destinations. Because of these stringent limitations on the tools, sometimes even a one-step transformation process had to be done in multiple steps. These convoluted approaches mandated the need to have unnecessary wastage in terms of manpower, as well as other computing resources. The main argument from the commercial ETL vendors all the time remained the same, one size doesn't fit all. So use *our* suite of tools instead of the point products available on the market. Many organizations got into vendor lock-in because of the profuse need to process data. Almost all the tools introduced before the year 2005 did not make use of the real power of the multi-core architecture of the computers if they supported running their tools on the commodity hardware. So, simple but voluminous data processing jobs took hours and sometimes even days to complete with these tools.

Spark became an instant hit in the market because of its ability to process a huge amount of data types and a growing number of data sources and data destinations. The most important and basic data abstraction Spark provides is the **resilient distributed dataset** (**RDD**). As discussed in the previous chapter, Spark supports distributed processing on a cluster of nodes. The moment there is a cluster of nodes, there is a good chance that when the data processing is going on, some of the nodes can die. When such failures happen, the framework should be capable of coming out of such failures. Spark is designed to do that and that is what the *resilient* part in the RDD signifies. If there is a huge amount of data to be processed and there are nodes available in the cluster, the framework should have the capability to split the big dataset into smaller chunks and distribute them to be processed on more than one node in a cluster, in parallel. Spark is capable of doing that and that is what the *distributed* part in the RDD signifies. In other words, Spark is designed from the ground

up to have its basic dataset abstraction capable of getting split into smaller pieces deterministically and distributed to more than one node in the cluster for parallel processing, while elegantly handling the failures in the nodes.

We will cover the following topics in this chapter:

- Functional programming with Spark
- Spark RDD
- Data transformations and actions
- Spark monitoring
- Spark programming basics
- Creating RDDs from files
- Spark libraries

Functional programming with Spark

The mutation of objects at run time, and the inability to get consistent results from a program or function because of the side effect that the program logic creates makes many applications very complex. If the functions in programming languages start behaving exactly like mathematical functions in such a way that the output of the function depends only on the inputs, that gives lots of predictability to applications. The computer programming paradigm giving lots of emphasis to the process of building such functions and other elements based on that, and using those functions just in the way that any other data types are being used, is popularly known as the functional programming paradigm. Out of the JVM-based programming languages, Scala is one of the most important ones that has very strong functional programming capabilities without losing any object orientation. Spark is written predominantly in Scala. Because of that itself, Spark has taken lots of very good concepts from Scala.

Understanding Spark RDD

The most important feature that Spark took from Scala is the ability to use functions as parameters to the Spark transformations and Spark actions. Quite often, the RDD in Spark behaves just like a collection object in Scala. Because of that, some of the data transformation method names of Scala collections are used in Spark RDD to do the same thing. This is a very neat approach and those who have expertise in Scala will find it very easy to program with RDDs. We will see a few important features in the following sections.

Spark RDD is immutable

There are some strong rules based on which an RDD is created. Once an RDD is created, intentionally or unintentionally, it cannot be changed. This gives another insight into the construction of an RDD. Because of that, when the nodes processing some part of an RDD die, the driver program can recreate those parts and assign the task of processing it to another node, ultimately, completing the data processing job successfully.

Since the RDD is immutable, splitting a big one to smaller ones, distributing them to various worker nodes for processing, and finally compiling the results to produce the final result can be done safely without worrying about the underlying data getting changed.

Spark RDD is distributable

If Spark is run in a cluster mode where there are multiple worker nodes available to take the tasks, all these nodes will have different execution contexts. The individual tasks are distributed and run on different JVMs. All these activities of a big RDD getting divided into smaller chunks, getting distributed for processing to the worker nodes, and finally, assembling the results back, are completely hidden from the users.

Spark has its own mechanism for recovering from the system faults and other forms of errors which occur during the data processing and hence this data abstraction is highly resilient.

Spark RDD lives in memory

Spark does keep all the RDDs in the memory as much as it can. Only in rare situations, where Spark is running out of memory or if the data size is growing beyond the capacity, is it written to disk. Most of the processing on RDD happens in the memory, and that is the reason why Spark is able to process the data at a lightning fast speed.

Spark RDD is strongly typed

Spark RDD can be created using any supported data types. These data types can be Scala/Java supported intrinsic data types or custom created data types such as your own classes. The biggest advantage coming out of this design decision is the freedom from runtime errors. If it is going to break because of a data type issue, it will break during compile time.

The following table captures the structure of an RDD containing tuples of a retail bank account data. It is of the type RDD[(string, string, string, double)]:

AccountNo	FirstName	LastName	AccountBalance
SB001	**John**	**Mathew**	**250.00**
SB002	Tracy	Mason	450.00
SB003	Paul	Thomson	560.00
SB004	Samantha	Grisham	650.00
SB005	John	Grove	1000.00

Suppose this RDD is going through a process to calculate the total amount of all these accounts in a cluster of three nodes, N1, N2, and N3; it can be split and distributed for something such as parallelizing the data processing. The following table contains the elements of the RDD[(string, string, string, double)] distributed to node N1 for processing:

AccountNo	FirstName	LastName	AccountBalance
SB001	**John**	**Mathew**	**250.00**
SB002	Tracy	Mason	450.00

The following table contains the elements of the RDD[(string, string, string, double)] distributed to node N2 for processing:

AccountNo	FirstName	LastName	AccountBalance
SB003	**Paul**	**Thomson**	**560.00**
SB004	Samantha	Grisham	650.00
SB005	John	Grove	1000.00

On node N1, the summation process happens and the result is returned to the Spark driver program. Similarly, on node N2, the summation process happens, the result is returned to the Spark driver program, and the final result is computed.

Spark has very deterministic rules on splitting a big RDD into smaller chunks for distribution to various nodes and because of that, even if something happens to, say, node N1, Spark knows how to recreate exactly the chunk that was lost in the node N1 and continue with the data processing operation by sending the same payload to node N3.

Figure 1 captures the essence of the process:
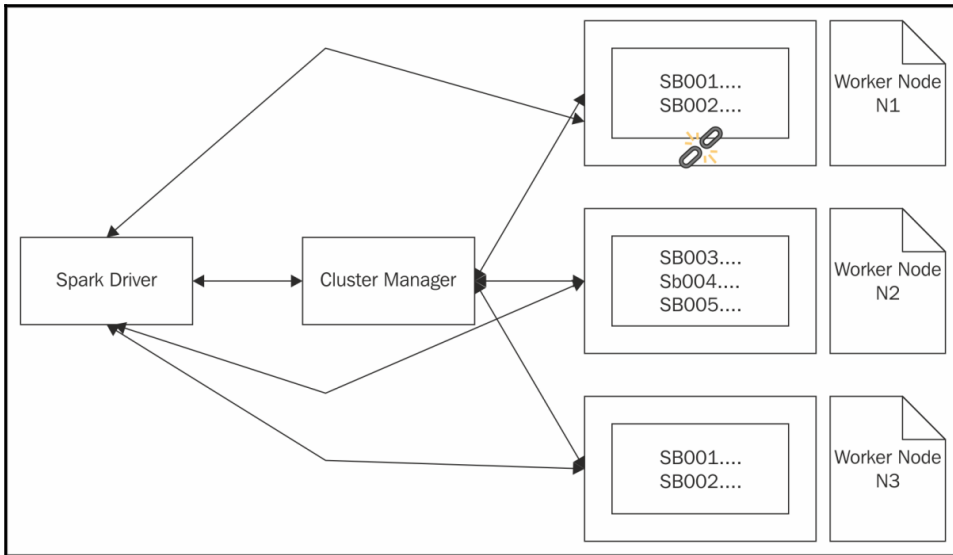
Figure 1

> **TIP**
>
> Spark does a lot of processing in its driver memory and in the executor memory on the cluster nodes. Spark has various parameters that can be configured and fine-tuned so that the required resources are made available before the processing starts.

Data transformations and actions with RDDs

Spark does the data processing using the RDDs. From the relevant data source such as text files and NoSQL data stores, data is read to form the RDDs. On such an RDD, various data transformations are performed and finally, the result is collected. To be precise, Spark comes with Spark transformations and Spark actions that act upon RDDs. Let us take the following RDD capturing a list of retail banking transactions, which is of the type RDD[(string, string, double)]:

AccountNo	TranNo	TranAmount
SB001	**TR001**	**250.00**
SB002	TR004	450.00

AccountNo	TranNo	TranAmount
SB003	TR010	120.00
SB001	TR012	-120.00
SB001	TR015	-10.00
SB003	TR020	100.00

To calculate the account level summary of the transactions from the RDD of the form
`(AccountNo,TranNo,TranAmount)`:

1. First it has to be transformed to the form of key-value pairs
   `(AccountNo, TranAmount)`, where `AccountNo` is the key but there will be
   multiple elements with the same key.
2. On this key, do a summation operation on `TranAmount`, resulting in another
   RDD of the form (AccountNo,TotalAmount),where every AccountNo will have
   only one element and TotalAmount is the sum of all the TranAmount for the
   given AccountNo.
3. Now sort the key-value pairs on the `AccountNo` and store the output.

In the whole process described, all are Spark transformations except the storing of the
output. Storing of the output is a **Spark action**. Spark does all these operations on a need-to-
do basis. Spark does not act when a Spark transformation is applied. The real act happens
when the first Spark action in the chain is called. Then it diligently applies all the preceding
Spark transformations in order, and then does the first encountered Spark action. This is
based on the concept called **Lazy Evaluation**.

> In the programming language context of declaring and using variables,
> *Lazy Evaluation* means that a variable is evaluated only when it is first used
> in the program.

Apart from this action of storing the output to disk, there are many other possible Spark
actions including, but not limited to, some of the ones given in the following list:

- Collecting all the contents in the resultant RDD to an array residing in the driver
  program
- Counting the number of elements in the RDD
- Counting the number of elements for each key in the RDD element

- Taking the first element in the RDD
- Taking a given number of elements from the RDD commonly used for top N reports
- Taking a sample of elements from the RDD
- Iterating through all the elements in the RDD

In this example, many transformations are done on various RDDs that get created on the fly till the process is completed. In other words, whenever a transformation is done on an RDD, a new RDD gets created. This is because RDDs are inherently immutable. These RDDs that are getting created at the end of each transformation can be saved for future reference, or they will go out of scope eventually.

To summarize, the process of creating one or more RDDs and applying transformations and actions on them is a very common usage pattern seen ubiquitously in Spark applications.

> The table referred in the preceding data transformation example contains the values in an RDD of type the RDD[(string, string, double)]. In this RDD, there are multiple elements, and each one is a tuple of the type (string, string, double). It is very common among programmers and the user community, for easy reference and conveying ideas, that the term `record` is being used to refer one to element in the RDD. In Spark RDD there is no concept of records, rows and columns. In other words the term `record` is mistakenly used synonymously to an element in the RDD, which may be a complex data type such as a tuple or a non-scalar data type. In this book, this practice is highly refrained to use the correct terms.

In Spark there are a good amount of Spark transformations available. These are really powerful because most of these take functions as input parameters to do the transformation. In other words, these transformations act on the RDD based on the functions that are defined and supplied by the user. This becomes even more powerful with Spark's uniform programming model. Whether the programming language of choice is Scala, Java, Python, or R, the way Spark transformations and Spark actions are used is similar. This lets the organizations choose their programming language of choice.

In Spark, even though the number of Spark actions are limited in number, they are really powerful, and users can write their own Spark actions if there is a need. There are many Spark connector programs that are available in the market, mainly to read and write data from various data stores. These connector programs are designed and developed either by the user community or by the data store vendors themselves to have connectivity to Spark. In addition to the available Spark actions, they may define their own actions to supplement existing sets of Spark actions. For example, the Spark Cassandra Connector is used to connect to Cassandra from Spark. This has an action `saveToCassandra`.

Monitoring with Spark

The previous chapter covered the details of the installation and development tool setup that is required for developing and running data processing applications using Spark. In most of the real-world applications, the Spark applications can become very complex with a really huge **directed acyclic graph** (**DAG**) of Spark transformations and Spark actions. Spark comes with really powerful monitoring tools for monitoring the jobs that are running in a given Spark ecosystem. The monitoring doesn't start automatically.

> Note that this is a completely optional step for running Spark applications. If enabled, it will give a very good insight into the way the Spark applications are run. Discretion has to be used to enable this in production environments, as it can affect the response time of the applications.

First of all, there are some configuration changes to be made. The event logging mechanism should be turned on. For this, take the following steps:

```
$ cd $SPARK_HOME
$ cd conf
$ cp spark-defaults.conf.template spark-defaults.conf
```

Once the preceding steps are completed, edit the newly created spark-defaults.conf file to have the following properties:

```
spark.eventLog.enabled          true
spark.eventLog.dir              <give a log directory location>
```

> Once the preceding steps are completed, make sure that the previously used log directory exists in the filesystem.

Apart from the preceding configuration file changes, there are many properties in that configuration file that can be changed to fine tune the Spark runtime. The most important among them that is used frequently is the Spark driver memory. If the applications are dealing with a huge amount of data, it is a good idea to customize this property spark.driver.memory to have a higher value. Then run the following commands to start the Spark master:

```
$ cd $SPARK_HOME
$ ./sbin/start-master.sh
```

Once the preceding steps are completed, make sure that the Spark web **user interface** (**UI**) is starting up by going to http://localhost:8080/. The assumption here is that there is

---

[38]

no other application running in the `8080` port. If for some reason, there is a need to run this application on a different port, the command line option `--webui-port <PORT>` can be used in the script while starting the web user interface.

The web UI should look something similar to that shown in Figure 2:

Worker Id	Address	State	Cores	Memory

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
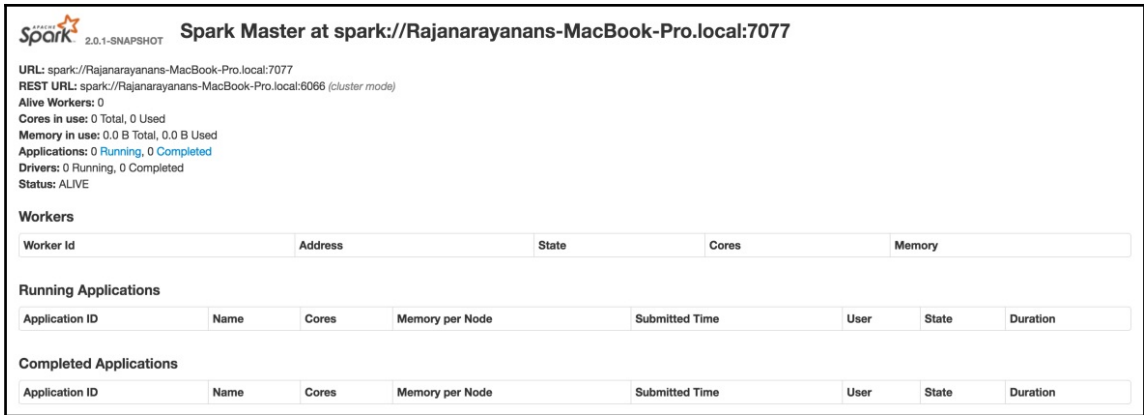
Figure 2

The most important information to be noted in the preceding figure is the fully-qualified Spark master URL (not the REST URL). It is going to be used again and again for many of the hands-on exercises that are going to be discussed in this book. The URL can change from system to system and the DNS settings. Also note that throughout this book, for all the hands-on exercises, Spark standalone deployment is used, which is the easiest among the deployments to get started with a single computer.

> These Spark application monitoring steps are given now to make the readers familiar with the toolset that Spark provides. Those who are familiar with these tools or those who are very confident of the application behavior need not need the help of these tools. But to understand the concepts, debugging, and some visualizations of the processes, these tools definitely provide immense help.

From the Spark web UI that is given in Figure 2, it can be noted that there are no worker nodes available to do any task, and there are no running applications. The following steps capture the instructions to start the worker nodes. Note how the Spark master URL is being used while starting the worker node:

```
$ cd $SPARK_HOME
$ ./sbin/start-slave.sh spark://Rajanarayanans-MacBook-Pro.local:7077
```

Once the worker node is started, in the Spark web UI, the newly started worker node is displayed. The
`$SPARK_HOME/conf/slaves.template` template captures the default worker nodes that will be started with the invocation of the preceding command.

> If additional worker nodes are required, copy the `slaves.template` file to name it to
> slaves and have the entries captured in there. When a spark-shell, pyspark, or sparkR
> is started, instructions can be given to it to use a given Spark master. This is useful when there is a need to run Spark applications or statements on a remote Spark cluster or against a given Spark master. If nothing is given, the Spark applications will run in the local mode.

```
$ cd $SPARK_HOME
$ ./bin/spark-shell --master spark://Rajanarayanans-MacBook-Pro.local:7077
```

The Spark web UI will look similar to that shown in Figure 3 once a worker node is started successfully. After this, if an application is run with the preceding Spark master URL, even that application's details will be displayed in the Spark web UI. A detailed coverage of the applications is to follow in this chapter. Use the following scripts to stop the workers and master processes:

```
$ cd $SPARK_HOME
$ ./sbin/stop-all.sh
```


Spark 2.0.1-SNAPSHOT   **Spark Master at spark://Rajanarayanans-MacBook-Pro.local:7077**

URL: spark://Rajanarayanans-MacBook-Pro.local:7077
REST URL: spark://Rajanarayanans-MacBook-Pro.local:6066 *(cluster mode)*
Alive Workers: 1
Cores in use: 8 Total, 0 Used
Memory in use: 7.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20160707203120-192.168.0.11-50405	192.168.0.11:50405	ALIVE	8 (0 Used)	7.0 GB (0.0 B Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Completed Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration

Figure 3

The basics of programming with Spark

Spark programming revolves around RDDs. In any Spark application, the input data to be processed is taken to create an appropriate RDD. To begin with, start with the most basic way of creating an RDD, which is from a list. The input data used for this `hello world` kind of application is a small collection of retail banking transactions. To explain the core concepts, only some very elementary data items have been picked up. The transaction records contain account numbers and transaction amounts.

> **TIP**
> In these use cases and all the upcoming use cases in the book, if the term record is used, that will be in the business or use case context.

The use cases selected for elucidating the Spark transformations and Spark actions here are given as follows:

1. The transaction records are coming as comma-separated values.
2. Filter out only the good transaction records from the list. The account number should start with `SB` and the transaction amount should be greater than zero.
3. Find all the high value transaction records with a transaction amount greater than 1000.
4. Find all the transaction records where the account number is bad.
5. Find all the transaction records where the transaction amount is less than or equal to zero.
6. Find a combined list of all the bad transaction records.
7. Find the total of all the transaction amounts.
8. Find the maximum of all the transaction amounts.
9. Find the minimum of all the transaction amounts.
10. Find all the good account numbers.

The approach that is going to be followed throughout the book for any application that is going to be developed begins with the Spark REPL for the appropriate language. Start the Scala REPL for Spark and make sure that it starts without any errors and the prompt is seen. For this application, we will enable monitoring to learn how to do that and use it along with the development process. Other than explicitly starting the Spark master and the slaves separately, Spark comes with a script that will start both of these together using a single script. Then, fire up the Scala REPL with the Spark master URL:

```
$ cd $SPARK_HOME
$ ./sbin/start-all.sh
```

```
$ ./bin/spark-shell --master spark://Rajanarayanans-MacBook-Pro.local:7077
```

At the Scala REPL prompt, try the following statements. The output of the statements is given in bold. Note that `scala>` is the Scala REPL prompt:

```
scala> val acTransList = Array("SB10001,1000", "SB10002,1200",
"SB10003,8000", "SB10004,400", "SB10005,300", "SB10006,10000",
"SB10007,500", "SB10008,56", "SB10009,30","SB10010,7000", "CR10001,7000",
"SB10002,-10")
acTransList: Array[String] = Array(SB10001,1000, SB10002,1200,
SB10003,8000, SB10004,400, SB10005,300, SB10006,10000, SB10007,500,
SB10008,56, SB10009,30, SB10010,7000, CR10001,7000, SB10002,-10)
scala> val acTransRDD = sc.parallelize(acTransList)
acTransRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at
parallelize at <console>:23
scala> val goodTransRecords = acTransRDD.filter(_.split(",")(1).toDouble >
0).filter(_.split(",")(0).startsWith("SB"))
goodTransRecords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at
filter at <console>:25
scala> val highValueTransRecords =
goodTransRecords.filter(_.split(",")(1).toDouble > 1000)
highValueTransRecords: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[3] at filter at <console>:27
scala> val badAmountLambda = (trans: String) =>
trans.split(",")(1).toDouble <= 0
badAmountLambda: String => Boolean = <function1>
scala> val badAcNoLambda = (trans: String) =>
trans.split(",")(0).startsWith("SB") == false
badAcNoLambda: String => Boolean = <function1>
scala> val badAmountRecords = acTransRDD.filter(badAmountLambda)
badAmountRecords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[4] at
filter at <console>:27
scala> val badAccountRecords = acTransRDD.filter(badAcNoLambda)
badAccountRecords: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[5]
at filter at <console>:27
scala> val badTransRecords  = badAmountRecords.union(badAccountRecords)
badTransRecords: org.apache.spark.rdd.RDD[String] = UnionRDD[6] at union at
<console>:33
```

All the preceding statements fall into one category except the first RDD creation and two function value definitions. They are all Spark transformations. Here is the step-by-step detail capturing what has been done so far:

- The value `acTransList` is the array containing the comma separated transaction records.

- The value `acTransRDD` is the RDD created out of the array where `sc` is the Spark context or the Spark driver and the RDD is created in a parallelized way so that the RDD elements can form a distributed dataset. In other words, an instruction is given to the Spark driver to form a parallel collection or RDD from the given collection of values.
- The value `goodTransRecords` is the RDD created from `acTransRDD` after filtering the conditions transaction amount is > 0 and the account number starts with `SB`.
- The value `highValueTransRecords` is the RDD created from `goodTransRecords` after filtering the conditions transaction amount is > 1000.
- The next two statements are storing the function definitions in a Scala value for easy reference later.

- The values `badAmountRecords` and `badAccountRecords` are RDDs created from `acTransRDD` to filter the bad records containing the wrong transaction amount and invalid account number, respectively.
- The value `badTransRecords` contains the union of the elements of both of the `badAmountRecords` and `badAccountRecords` RDDs.

The Spark web UI for this application so far will not show anything at this point because only Spark transformations have been executed. The real activity will start only after the first Spark action is executed.

The following statements are the continuation of the already executed statements:

```
scala> acTransRDD.collect()
res0: Array[String] = Array(SB10001,1000, SB10002,1200, SB10003,8000,
SB10004,400, SB10005,300, SB10006,10000, SB10007,500, SB10008,56,
SB10009,30, SB10010,7000, CR10001,7000, SB10002,-10)
scala> goodTransRecords.collect()
res1: Array[String] = Array(SB10001,1000, SB10002,1200, SB10003,8000,
SB10004,400, SB10005,300, SB10006,10000, SB10007,500, SB10008,56,
SB10009,30, SB10010,7000)
scala> highValueTransRecords.collect()
res2: Array[String] = Array(SB10002,1200, SB10003,8000, SB10006,10000,
SB10010,7000)
scala> badAccountRecords.collect()
res3: Array[String] = Array(CR10001,7000)
scala> badAmountRecords.collect()
res4: Array[String] = Array(SB10002,-10)
scala> badTransRecords.collect()
res5: Array[String] = Array(SB10002,-10, CR10001,7000)
```

All the preceding statements did one thing, which is execute a Spark action on the RDDs *defined* earlier. All the evaluations of the RDDs happened only when a Spark action was called on those RDDs. The following statements are doing some of the calculations on the RDDs:

```
scala> val sumAmount = goodTransRecords.map(trans =>
trans.split(",")(1).toDouble).reduce(_ + _)
sumAmount: Double = 28486.0
scala> val maxAmount = goodTransRecords.map(trans =>
trans.split(",")(1).toDouble).reduce((a, b) => if (a > b) a else b)
maxAmount: Double = 10000.0
scala> val minAmount = goodTransRecords.map(trans =>
trans.split(",")(1).toDouble).reduce((a, b) => if (a < b) a else b)
minAmount: Double = 30.0
```

The preceding numbers calculated the sum, maximum and minimum, of all transaction amounts from the good records. In all the preceding transformations, the transaction records are processed one at a time. From those records, the account number and transaction amount are extracted and processed. It was done like that because the use case requirement was like that. Now the comma-separated values in each transaction record are split without looking at whether it is an account number or a transaction amount. The resulting RDD will contain a collection with all these mixed up. Out of that, if the elements starting with SB are picked up, that will result in good account numbers. The following statements are going to do that:

```
scala> val combineAllElements = acTransRDD.flatMap(trans =>
trans.split(","))
combineAllElements: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[10]
at flatMap at <console>:25
scala> val allGoodAccountNos =
combineAllElements.filter(_.startsWith("SB"))
allGoodAccountNos: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11]
at filter at <console>:27
scala> combineAllElements.collect()
res10: Array[String] = Array(SB10001, 1000, SB10002, 1200, SB10003, 8000,
SB10004, 400, SB10005, 300, SB10006, 10000, SB10007, 500, SB10008, 56,
SB10009, 30, SB10010, 7000, CR10001, 7000, SB10002, -10)
scala> allGoodAccountNos.distinct().collect()
res14: Array[String] = Array(SB10006, SB10010, SB10007, SB10008, SB10009,
SB10001, SB10002, SB10003, SB10004, SB10005)
```

Now at this point, if the Spark web UI is opened, unlike what is seen in Figure 3, one difference can be noticed. Since some Spark actions have been done, an application entry will show up. Since the Scala REPL of Spark is still running, it is shown in the list of applications that are still running. The following Figure 4 captures that:
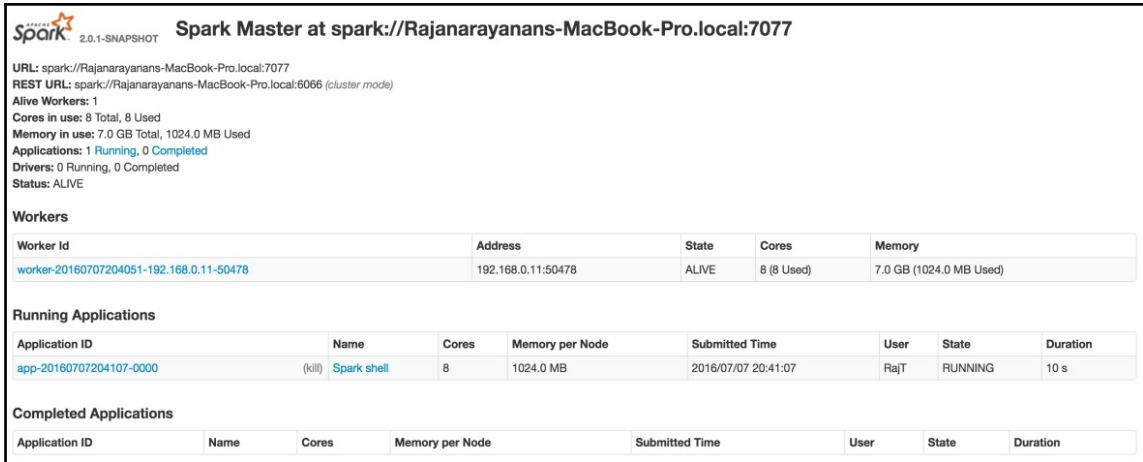
Figure 4

Navigate by clicking on the application ID to see all the metrics related to the running applications including the DAG visualizations and many more.

These statements covered all the use cases discussed, and it is worth going through the Spark transformations covered so far. These are some of the basic but very important transformations that will be used in most of the applications again and again:

Spark transformation	What it does
`filter(fn)`	**Iterates through all the elements of the RDD, applies the function that is passed, and picks up the elements that return true as evaluated by the function on the element.**
`map(fn)`	Iterates through all the elements of the RDD, applies the function that is passed, and picks up the output returned by the function.
`flatMap(fn)`	Iterates through all the elements of the RDD, applies the function that is passed, and picks up the output returned by the function. The big difference here as compared to the Spark transformation `map(fn)` is that the function acts on a single element and returns a flat collection of elements. For example, it takes one banking transaction record and splits it into multiple fields, resulting in a collection from a single element.
`union(other)`	Takes the union of all the elements of this RDD and the other RDD.

It is also worth going through the Spark actions covered so far. These are some of the basic ones, but more actions will be covered in due course.

Spark action	What it does
`collect()`	**Collects all the elements in the RDD to an array in the Spark driver.**
`reduce(fn)`	Applies the function fn on all the elements of the RDD and the final result is calculated as defined by the function. It should be a function that takes two parameters and returns one, which is also commutative and associative.
`foreach(fn)`	Applies the function fn on all the elements of the RDD. This is mainly used for side effects. The Spark transformation `map(fn)` applies the function to all the elements of the RDD and returns another RDD. But the `foreach(fn)` Spark transformation does not return an RDD. For example, `foreach(println)` will take each element from the RDD and print it onto the console. Even though it is not used in the use cases covered here, it is worth mentioning.

The next step in the Spark learning process is to try the statements in the Python REPL, covering exactly the same use case. The variable definitions have been maintained as similar as possible in both the languages to have easy assimilation of ideas. There may be minor variations in the way they are used here as compared to the Scala way; conceptually, it is independent of the language of choice.

Start the Python REPL for Spark and make sure that it starts without any errors and the prompt is seen. While playing around with Scala code, the monitoring was already enabled. Now fire up the Python REPL with the Spark master URL:

```
$ cd $SPARK_HOME
$ ./bin/pyspark --master spark://Rajanarayanans-MacBook-Pro.local:7077
```

At the Python REPL prompt, try the following statements. The output of the statements is given in bold. Note that >>> is the Python REPL prompt:

```
>>> from decimal import Decimal
>>> acTransList = ["SB10001,1000", "SB10002,1200", "SB10003,8000",
"SB10004,400", "SB10005,300", "SB10006,10000", "SB10007,500", "SB10008,56",
"SB10009,30","SB10010,7000", "CR10001,7000", "SB10002,-10"]
>>> acTransRDD = sc.parallelize(acTransList)
>>> goodTransRecords = acTransRDD.filter(lambda trans:
Decimal(trans.split(",")[1]) > 0).filter(lambda trans:
(trans.split(",")[0]).startswith('SB') == True)
>>> highValueTransRecords = goodTransRecords.filter(lambda trans:
Decimal(trans.split(",")[1]) > 1000)
>>> badAmountLambda = lambda trans: Decimal(trans.split(",")[1]) <= 0
>>> badAcNoLambda = lambda trans: (trans.split(",")[0]).startswith('SB') ==
False
```

```
>>> badAmountRecords = acTransRDD.filter(badAmountLambda)
>>> badAccountRecords = acTransRDD.filter(badAcNoLambda)
>>> badTransRecords  = badAmountRecords.union(badAccountRecords)
>>> acTransRDD.collect()
['SB10001,1000', 'SB10002,1200', 'SB10003,8000', 'SB10004,400',
'SB10005,300', 'SB10006,10000', 'SB10007,500', 'SB10008,56', 'SB10009,30',
'SB10010,7000', 'CR10001,7000', 'SB10002,-10']
>>> goodTransRecords.collect()
['SB10001,1000', 'SB10002,1200', 'SB10003,8000', 'SB10004,400',
'SB10005,300', 'SB10006,10000', 'SB10007,500', 'SB10008,56', 'SB10009,30',
'SB10010,7000']
>>> highValueTransRecords.collect()
['SB10002,1200', 'SB10003,8000', 'SB10006,10000', 'SB10010,7000']
>>> badAccountRecords.collect()
['CR10001,7000']
>>> badAmountRecords.collect()
['SB10002,-10']
>>> badTransRecords.collect()
['SB10002,-10', 'CR10001,7000']
>>> sumAmounts = goodTransRecords.map(lambda trans:
Decimal(trans.split(",")[1])).reduce(lambda a,b : a+b)
>>> sumAmounts
Decimal('28486')
>>> maxAmount = goodTransRecords.map(lambda trans:
Decimal(trans.split(",")[1])).reduce(lambda a,b : a if a > b else b)
>>> maxAmount
Decimal('10000')
>>> minAmount = goodTransRecords.map(lambda trans:
Decimal(trans.split(",")[1])).reduce(lambda a,b : a if a < b else b)
>>> minAmount
Decimal('30')
>>> combineAllElements = acTransRDD.flatMap(lambda trans: trans.split(","))
>>> combineAllElements.collect()
['SB10001', '1000', 'SB10002', '1200', 'SB10003', '8000', 'SB10004', '400',
'SB10005', '300', 'SB10006', '10000', 'SB10007', '500', 'SB10008', '56',
'SB10009', '30', 'SB10010', '7000', 'CR10001', '7000', 'SB10002', '-10']
>>> allGoodAccountNos = combineAllElements.filter(lambda trans:
trans.startswith('SB') == True)
>>> allGoodAccountNos.distinct().collect()
['SB10005', 'SB10006', 'SB10008', 'SB10002', 'SB10003', 'SB10009',
'SB10010', 'SB10004', 'SB10001', 'SB10007']
```

The real power of the uniform programming model of Spark is very clearly visible if both the Scala and Python code sets are compared. The Spark transformations and Spark actions are the same in both the language implementations. The way functions are passed into these are different because of the programming language syntax differences.

Before running the Python REPL for Spark, the Scala REPL was closed and this was done on purpose. Then the Spark web UI should look something similar to that shown in Figure 5. Since the Scala REPL was closed, that is getting listed under the completed applications list. Since the Python REPL is still open, that is getting listed under the running applications list. Note the application names of both the Scala REPL and Python REPL of Spark in the Spark web UI. These are standard names. When custom applications are run from files, there are ways to assign custom names while defining the Spark context object for the applications to facilitate monitoring of the applications and for logging purposes. These details will be covered later in this chapter.

It is a good idea to spend time with the Spark web UI, getting familiar with all the metrics that are being captured, and how the DAG visualization is given in the UI. It will help a lot while debugging complex Spark applications.
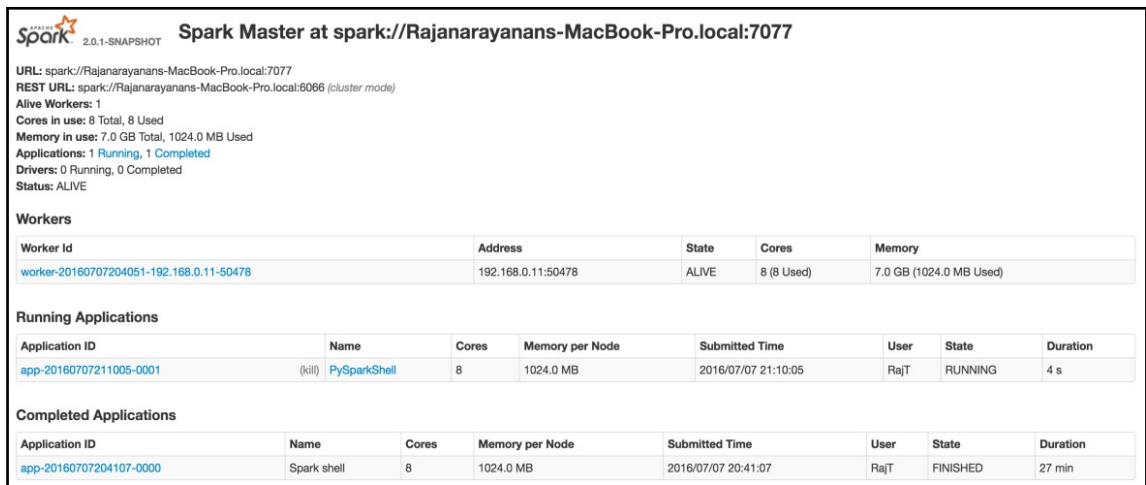
Figure 5

MapReduce

Since day one, Spark has been placed as the replacement for Hadoop MapReduce programs. In general, data processing jobs are done in MapReduce style if that job can be divided into multiple tasks and they can be executed in parallel, and the final results can be computed after collecting the results from all these distributed pieces. Unlike Hadoop MapReduce, Spark can do this even if the DAG of activities is more than the two stages, such as Map and Reduce. Spark is designed to do that and that is one of the biggest value propositions that Spark highlights.

This section is going to continue with the same retail banking application and pick up some of the use cases that are ideal candidates for the MapReduce kind of data processing.

The use cases selected for elucidating the MapReduce kind of data processing here are given as follows:

1. The retail banking transaction records come with account numbers and the transaction amounts in comma-separated strings.
2. Pair the transactions to have key/value pairs such as (`AccNo`, `TranAmount`).
3. Find an account level summary of all the transactions to get the account balance.

At the Scala REPL prompt, try the following statements:

```scala
scala> val acTransList = Array("SB10001,1000", "SB10002,1200",
"SB10001,8000", "SB10002,400", "SB10003,300", "SB10001,10000",
"SB10004,500", "SB10005,56", "SB10003,30","SB10002,7000", "SB10001,-100",
"SB10002,-10")
acTransList: Array[String] = Array(SB10001,1000, SB10002,1200,
SB10001,8000, SB10002,400, SB10003,300, SB10001,10000, SB10004,500,
SB10005,56, SB10003,30, SB10002,7000, SB10001,-100, SB10002,-10)
scala> val acTransRDD = sc.parallelize(acTransList)
acTransRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at
parallelize at <console>:23
scala> val acKeyVal = acTransRDD.map(trans => (trans.split(",")(0),
trans.split(",")(1).toDouble))
acKeyVal: org.apache.spark.rdd.RDD[(String, Double)] = MapPartitionsRDD[1]
at map at <console>:25
scala> val accSummary = acKeyVal.reduceByKey(_ + _).sortByKey()
accSummary: org.apache.spark.rdd.RDD[(String, Double)] = ShuffledRDD[5] at
sortByKey at <console>:27
scala> accSummary.collect()
res0: Array[(String, Double)] = Array((SB10001,18900.0), (SB10002,8590.0),
(SB10003,330.0), (SB10004,500.0), (SB10005,56.0))
```

Here is the step-by-step detail capturing what has been done so far:

1. The value `acTransList` is the array containing the comma-separated transaction records.
2. The value `acTransRDD` is the RDD created out of the array, where sc is the Spark context or the Spark driver and the RDD is created in a parallelized way so that the RDD elements can form a distributed dataset.
3. Transform the `acTransRDD` to `acKeyVal` to have key-value pairs of the form (K,V), where the account number is chosen as the key. In this set of elements in the RDD, there will be multiple elements with the same key.

4. In the next step, the key-value pairs are grouped by the key and a reduction function has been passed, which will add the transaction amount to form key-value pairs containing one element for a specific key in the RDD and the total of all the amounts for the same key. Then sort the elements on the key before producing the final result.
5. Collect the elements to an array at the driver level.

Assuming that the RDD `acKeyVal` is partitioned into two parts and distributed to a cluster for processing, Figure 6 captures the essence of the processing:
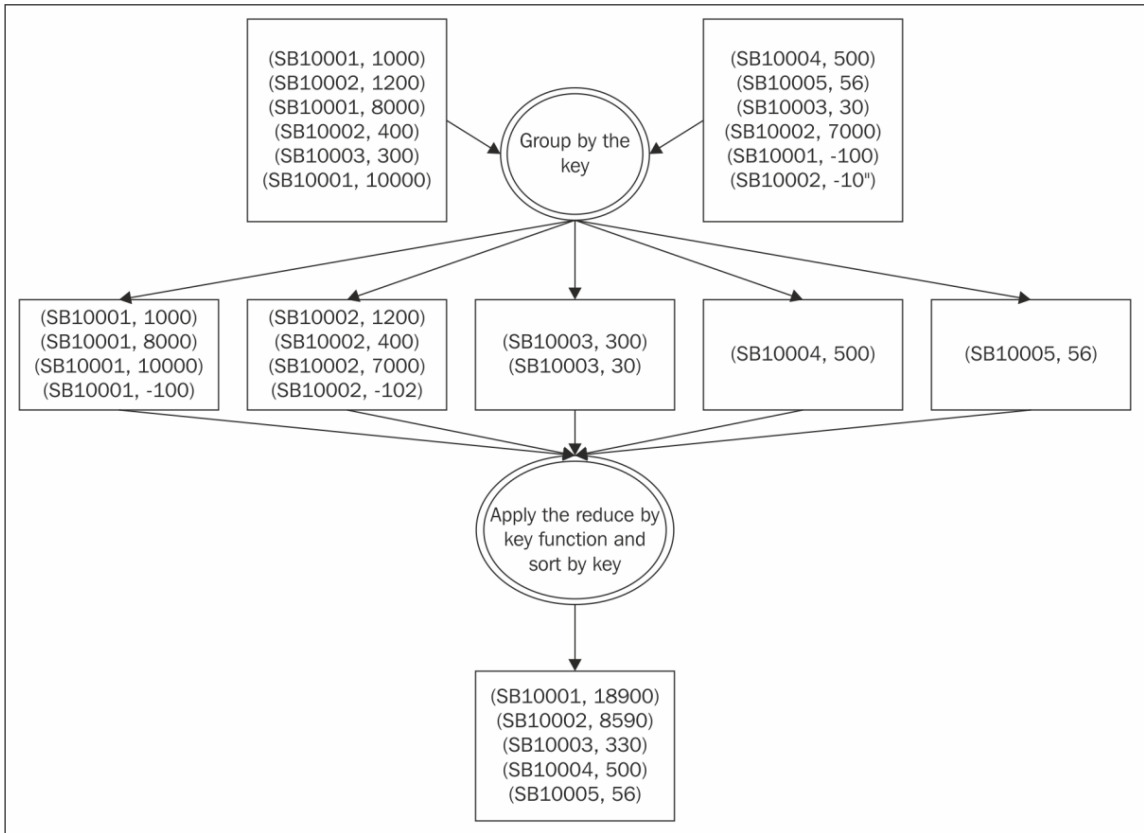
Figure 6

The following table captures the Spark actions that are introduced in this use case:

Spark action	What it does?
reduceByKey(fn,[noOfTasks])	**Applies the function fn on the RDD of the form (K,V) and is reduced to remove duplicate keys and apply the function passed as the parameter to be acted on the values at the key level.**
sortByKey([ascending], [numTasks])	Sorts the RDD elements if the RDD is of the form (K,V) by its key K

The reduceByKey action deserves a special mention. In Figure 6, the grouping of the elements by the key is a well-known operation. But in the next step, for the same key, the function passed to as a parameter takes two parameters and returns one. It is not very intuitive to get this right and you may be wondering from where the two inputs are coming while iterating through the values of the (K,V) pair for each key. This behavior takes the concept from the Scala collection method reduceLeft. The following Figure 7, with the values of the key **SB10001** doing the reduceByKey(_ + _) operation, is an attempt to explain the concept. This is just for the elucidation purposes of this example and the actual Spark implementation to do the same may be different:
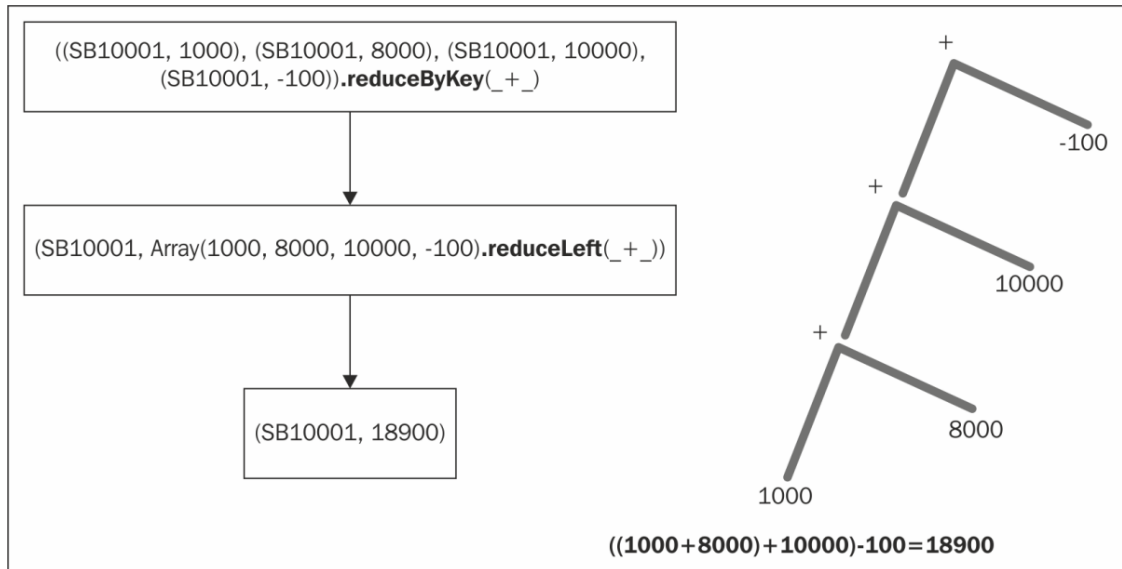
Figure 7

On the right-hand side of Figure 7, the `reduceLeft` operation of the Scala collection method is illustrated. That is an attempt to give some insight into from where the two parameters are coming for the `reduceLeft` function. As a matter of fact, many of the transformations that are being used on Spark RDD are adapted from Scala collection methods.

At the Python REPL prompt, try the following statements:

```
>>> from decimal import Decimal
>>> acTransList = ["SB10001,1000", "SB10002,1200", "SB10001,8000",
"SB10002,400", "SB10003,300", "SB10001,10000", "SB10004,500", "SB10005,56",
"SB10003,30","SB10002,7000", "SB10001,-100", "SB10002,-10"]
>>> acTransRDD = sc.parallelize(acTransList)
>>> acKeyVal = acTransRDD.map(lambda trans:
(trans.split(",")[0],Decimal(trans.split(",")[1])))
>>> accSummary = acKeyVal.reduceByKey(lambda a,b : a+b).sortByKey()
>>> accSummary.collect()
[('SB10001', Decimal('18900')), ('SB10002', Decimal('8590')), ('SB10003',
Decimal('330')), ('SB10004', Decimal('500')), ('SB10005', Decimal('56'))]
```

The `reduceByKey` took an input parameter, which is a function. Similar to this, there is another transformation that does the key-based operation in a slightly different way. It is `groupByKey()`. This gathers all the values of a given key and forms the list of values from all the individual elements.

If there is a need to do multiple levels of processing with the same value elements as a collection for each key, this is the suitable transformation. In other words, if there are many (K,V) pairs, this transformation returns (K, Iterable<V>) for each key.

> **TIP**
>
> The only thing the developer needs to be cognizant about is to make sure that the number of such (K,V) pairs is not really huge so that the operation doesn't create performance problems. There is no hard and fast rule to find this out and it rather depends on the use case.

In all the preceding code snippets, for extracting account numbers or any other field from the comma-separated transaction record, split(`,`) is used multiple times within the `map()` transformation. This is to demonstrate the use of array elements within `map()`, or any other transformation or method. A better way of extracting the fields of the transaction record is to transform them as a tuple containing the required fields and then use the fields from the tuple to employ them in some of the following code snippets. In this way, there is no need to call split (`,`) repeatedly for each field extraction.

Joins

In the **Relational Database Management Systems** (**RDBMS**) world, joining multiple tables rows based on a key is a very common practice. When it comes to the NoSQL data stores, joining multiple tables became a real problem because many of the NoSQL data stores don't have support for the table joins. In the NoSQL world, redundancy is allowed. Whether a technology supports table joins or not, business use cases mandate joins of datasets based on keys all the time. Because of this, it is imperative to have the joins done in a batch mode in many of the use cases.

Spark provides transformations to join multiple RDDs based on a key. This supports many use cases. These days there are many NoSQL data stores having connectors to talk to Spark. When working with such data stores, it is very simple to construct RDDs of data from multiple tables, do the join from Spark, and store the results back into the data stores in batch mode or even in near-to-real-time mode. Spark transformations are available for left outer join and right outer join, as well as full outer join.

The use cases selected for elucidating the join of multiple datasets using a key are given as follows.

The first dataset contains a retail banking master records summary with an account number, first name, and last name. The second dataset contains the retail banking account balance with an account number, and balance amount. The key on both of the datasets is the account number. Join the two datasets and create one dataset containing the account number, full name, and balance amount.

At the Scala REPL prompt, try the following statements:

```
scala> val acMasterList = Array("SB10001,Roger,Federer",
"SB10002,Pete,Sampras", "SB10003,Rafael,Nadal", "SB10004,Boris,Becker",
"SB10005,Ivan,Lendl")
acMasterList: Array[String] = Array(SB10001,Roger,Federer,
SB10002,Pete,Sampras, SB10003,Rafel,Nadal, SB10004,Boris,Becker,
SB10005,Ivan,Lendl)
scala> val acBalList = Array("SB10001,50000", "SB10002,12000",
"SB10003,3000", "SB10004,8500", "SB10005,5000")
acBalList: Array[String] = Array(SB10001,50000, SB10002,12000,
SB10003,3000, SB10004,8500, SB10005,5000)
scala> val acMasterRDD = sc.parallelize(acMasterList)
acMasterRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at
parallelize at <console>:23
scala> val acBalRDD = sc.parallelize(acBalList)
acBalRDD: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[1] at
parallelize at <console>:23
scala> val acMasterTuples = acMasterRDD.map(master =>
```

```
master.split(",")).map(masterList => (masterList(0), masterList(1) + " " +
masterList(2)))
acMasterTuples: org.apache.spark.rdd.RDD[(String, String)] =
MapPartitionsRDD[3] at map at <console>:25
scala> val acBalTuples = acBalRDD.map(trans =>
trans.split(",")).map(transList => (transList(0), transList(1)))
acBalTuples: org.apache.spark.rdd.RDD[(String, String)] =
MapPartitionsRDD[5] at map at <console>:25
scala> val acJoinTuples =
acMasterTuples.join(acBalTuples).sortByKey().map{case (accno, (name,
amount)) => (accno, name,amount)}
acJoinTuples: org.apache.spark.rdd.RDD[(String, String, String)] =
MapPartitionsRDD[12] at map at <console>:33
scala> acJoinTuples.collect()
res0: Array[(String, String, String)] = Array((SB10001,Roger
Federer,50000), (SB10002,Pete Sampras,12000), (SB10003,Rafael Nadal,3000),
(SB10004,Boris Becker,8500), (SB10005,Ivan Lendl,5000))
```

All the statements given previously must be familiar by now, except the Spark transformation
join. Similar to this transformation, the `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin` are also available with the same usage pattern:

Spark transformation	What it does
`join(other, [numTasks])`	**Joins this RDD with the other RDD, and the elements are joined together based on the key. Suppose the original RDD is of the form (K,V1) and the second RDD is of the form (K,V2), then the join operation will produce tuples of the form (K, (V1,V2)) with all the pairs of each key.**

At the Python REPL prompt, try the following statements:

```
>>> acMasterList = ["SB10001,Roger,Federer", "SB10002,Pete,Sampras",
"SB10003,Rafael,Nadal", "SB10004,Boris,Becker", "SB10005,Ivan,Lendl"]
>>> acBalList = ["SB10001,50000", "SB10002,12000", "SB10003,3000",
"SB10004,8500", "SB10005,5000"]
>>> acMasterRDD = sc.parallelize(acMasterList)
>>> acBalRDD = sc.parallelize(acBalList)
>>> acMasterTuples = acMasterRDD.map(lambda master:
master.split(",")).map(lambda masterList: (masterList[0], masterList[1] + "
" + masterList[2]))
>>> acBalTuples = acBalRDD.map(lambda trans: trans.split(",")).map(lambda
transList: (transList[0], transList[1]))
>>> acJoinTuples = acMasterTuples.join(acBalTuples).sortByKey().map(lambda
tran: (tran[0], tran[1][0],tran[1][1]))
>>> acJoinTuples.collect()
```

```
[('SB10001', 'Roger Federer', '50000'), ('SB10002', 'Pete Sampras',
'12000'), ('SB10003', 'Rafael Nadal', '3000'), ('SB10004', 'Boris Becker',
'8500'), ('SB10005', 'Ivan Lendl', '5000')]
```

More actions

So far, the focus was mainly on Spark transformations. Spark actions are also important. To get insight into some more important Spark actions, take the following use cases, continuing from where it was stopped in the preceding section's use cases:

- From the list containing account numbers, names, and account balances, get the one that has the highest account balance
- From the list containing account numbers, names, and account balances, get the top three having the highest account balance
- Count the number of balance transaction records at an account level
- Count the total number of balance transaction records
- Print the name and account balance of all the accounts
- Calculate the total of the account balance

> **TIP**
>
> It is a very common requirement to iterate through the elements in a collection, do some mathematical calculation on each of the elements, and at the end of it, use the result. The RDD is partitioned and distributed across worker nodes. If any normal variable is used for storing the cumulative result while iterating through the RDD elements, it may not yield the correct result. In such situations, instead of using regular variables, use Spark provided accumulators.

At the Scala REPL prompt, try the following statements:

```
scala> val acNameAndBalance = acJoinTuples.map{case (accno, name,amount) =>
(name,amount)}
acNameAndBalance: org.apache.spark.rdd.RDD[(String, String)] =
MapPartitionsRDD[46] at map at <console>:35
scala> val acTuplesByAmount = acBalTuples.map{case (accno, amount) =>
(amount.toDouble, accno)}.sortByKey(false)
acTuplesByAmount: org.apache.spark.rdd.RDD[(Double, String)] =
ShuffledRDD[50] at sortByKey at <console>:27
scala> acTuplesByAmount.first()
res19: (Double, String) = (50000.0,SB10001)
scala> acTuplesByAmount.take(3)
res20: Array[(Double, String)] = Array((50000.0,SB10001),
(12000.0,SB10002), (8500.0,SB10004))
scala> acBalTuples.countByKey()
```

```
res21: scala.collection.Map[String,Long] = Map(SB10001 -> 1, SB10005 -> 1,
SB10004 -> 1, SB10002 -> 1, SB10003 -> 1)
scala> acBalTuples.count()
res22: Long = 5
scala> acNameAndBalance.foreach(println)
(Boris Becker,8500)
(Rafel Nadal,3000)
(Roger Federer,50000)
(Pete Sampras,12000)
(Ivan Lendl,5000)
scala> val balanceTotal = sc.accumulator(0.0, "Account Balance Total")
balanceTotal: org.apache.spark.Accumulator[Double] = 0.0
scala> acBalTuples.map{case (accno, amount) => amount.toDouble}.foreach(bal
=> balanceTotal += bal)
scala> balanceTotal.value
res8: Double = 78500.0)
```

The following table captures the Spark actions that are introduced in this use case:

Spark action	What it does
`first()`	**Returns the first element in the RDD.**
`take(n)`	Returns an array of the first `n` elements from the RDD.
`countByKey()`	Returns the count of elements by the key. If the RDD contains (K,V) pairs, this will return a dictionary of (`K, numOfValues`).
`count()`	Returns the number of elements in the RDD.
`foreach(fn)`	Applies the function fn to each element in the RDD. In the preceding use case, Spark Accumulator is being used with `foreach(fn)`.

At the Python REPL prompt, try the following statements:

```
>>> acNameAndBalance = acJoinTuples.map(lambda tran: (tran[1],tran[2]))
>>> acTuplesByAmount = acBalTuples.map(lambda tran: (Decimal(tran[1]),
tran[0])).sortByKey(False)
>>> acTuplesByAmount.first()
(Decimal('50000'), 'SB10001')
>>> acTuplesByAmount.take(3)
[(Decimal('50000'), 'SB10001'), (Decimal('12000'), 'SB10002'),
(Decimal('8500'), 'SB10004')]
>>> acBalTuples.countByKey()
defaultdict(<class 'int'>, {'SB10005': 1, 'SB10002': 1, 'SB10003': 1,
'SB10004': 1, 'SB10001': 1})
>>> acBalTuples.count()
5
>>> acNameAndBalance.foreach(print)
```

```
('Pete Sampras', '12000')
('Roger Federer', '50000')
('Rafael Nadal', '3000')
('Boris Becker', '8500')
('Ivan Lendl', '5000')
>>> balanceTotal = sc.accumulator(0.0)
>>> balanceTotal.value 0.0
>>> acBalTuples.foreach(lambda bals: balanceTotal.add(float(bals[1])))
>>> balanceTotal.value
78500.0
```

Creating RDDs from files

So far, the focus of the discussion was on the RDD functionality and programming with RDDs. In all the preceding use cases, the RDD creation was done from the collection objects. But in the real-world use cases, the data will come from files stored in the local filesystems, and HDFS. Quite often, the data will come from NoSQL data stores such as Cassandra. It is possible to create RDDs by reading the contents from these data sources. Once RDD is created, then all the operations are uniform, as given in the preceding use cases. The data files coming out of the filesystems may be fixed width, comma-separated, or any other format. But the common pattern used for reading such data files is to read the data line by line and split the line to have the necessary separation of data items. In the case of data coming from other sources, the appropriate Spark connector program is to be used and the appropriate API for reading data is to be used.

Many third-party libraries are available to read the contents from various types of text files. For example, the Spark CSV library available from GitHub is a very useful one for creating RDDs from CSV files.

The following table captures the way text files are read from various sources, such as local filesystems, HDFS, and so on. As discussed earlier, the processing of the text file is up to the use case requirements:

File location	RDD creation	What it does
Local filesystem	```val textFile = sc.textFile("README.md")```	**Creates an RDD by reading the contents of the file named** ```README.md``` **from the directory from where the Spark shell is invoked. Here, the RDD is of the type RDD[string] and the elements will be lines from the file.**
HDFS	```val textFile = sc.textFile ("hdfs://<location in HDFS>")```	Creates an RDD by reading the contents of the file specified in the HDFS URL

The most important aspect while reading the files from the local filesystem is that the file should be available in all the nodes of the Spark worker nodes. Apart from these two file locations given in the preceding table, any supported filesystem URI may be used.

Just like reading the contents from files in various filesystems, it is also possible to write the RDD onto files using the ```saveAsTextFile```(path) Spark action.

> All the Spark application use cases discussed here are run on the appropriate language's REPL of Spark. When writing applications, they will be written in proper source code files. In the case of Scala and Java, the application code files have to be compiled, packaged, and run with proper library dependencies, and are typically built using maven or sbt. This will be covered in detail when designing data processing applications using Spark, in the last chapter of this book.

Understanding the Spark library stack

Spark comes with a core data processing engine and a stack of libraries working on top of the core engine. It is very important to understand the concept of stacking libraries on top of the core framework.

All these libraries that are making use of the services provided by the core framework support the data abstractions offered by the core framework and much more. Before Spark came onto market, there were lots of independent open source products doing what the library stack in discussion here is now doing. The biggest disadvantage with these point products was their interoperability. They don't stack together well. They were implemented in different programming languages. The programming language of choice supported by

these products, and the lack of uniformity in the APIs exposed by these products, were really challenging to get one application done with two or more such products. That is the relevance of the stack of libraries that work on top of Spark. They all work together with the same programming model. This helps organizations to standardize on the data processing toolset without vendorlock-in.

Spark comes with the following stack of domain-specific libraries, and Figure 8 gives a comprehensive picture of the whole ecosystem as seen by a developer:

- **Spark SQL**
- **Spark Streaming**
- **Spark MLlib**
- **Spark GraphX**
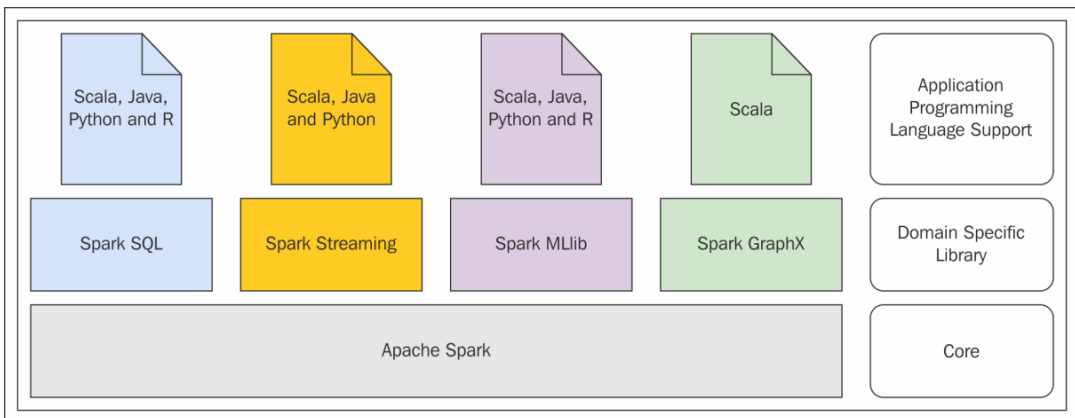
Figure 8

In any organization, structured data is still very widely used. The most ubiquitous data access mechanism with structured data is SQL. Spark SQL provides the capability to write SQL-like queries on top of the structured data abstraction called the DataFrame API. DataFrame and SQL go very well and support data coming from various sources, such as Hive, Avro, Parquet, JSON, and many more. Once the data is loaded into the Spark context, they can be operated as if they are all coming from the same source. In other words, if required, SQL-like queries can be used to join data coming from different sources, such as Hive and JSON. Another big advantage that Spark SQL and the DataFrame API bring onto the developers table is the ease of use and no need-to-know functional programming methods, which is a requirement to do programming with RDDs.

Using Spark SQL and the DataFrame API, data can be read from various data sources and processed as if it is all coming from a unified source. Spark transformations and Spark actions support uniform programming interfaces. So the data source unification, API unification, and ability to use multiple programming languages to write data processing applications help the organizations to standardize on one data processing framework.

The data ingestion into the organizational data sinks is increasing every day. At the same time, the velocity at which data is getting ingested is also increasing. Spark Streaming provides the library to process the data that is ingested from various sources at a very high velocity.

In the past, data scientists had the challenge of building their own implementations of the machine learning algorithms and utilities in their programming language of choice. Quite often, such programming languages don't interoperate with the data processing toolset of the organization. Spark MLlib provides the unification process, where it comes with a lot of machine learning algorithms and utilities working on top of the Spark data processing engine.

The IoT applications, especially the social media applications, mandated the need to have data processing capabilities where the data fits into a graph-like structure. For example, the connections in LinkedIn, relationship between friends in Facebook, workflow applications, and many such use cases, make use of the graph abstraction extensively. Using the graph to do various computations requires very high data processing capabilities and employment of sophisticated algorithms. Spark GraphX library comes with an API for graphs and makes use of Spark's parallel computing paradigm.

There are many Spark libraries available that are developed by the community for various purposes. Many such third-party library packages are featured in the site `http://spark-packages.org/`. The number of packages is growing day by day as the Spark user community is growing. When developing data processing applications in Spark, if there is a need to have a domain-specific library, it would be a good idea to check this site first and see whether anybody has already developed it.

Reference

For more information plese visit:`https://github.com/databricks/spark-csv`

Summary

This chapter discussed the basic programming model of Spark with its primary dataset abstraction RDDs. The creation of RDDs from various data sources, and processing of the data in RDDs using Spark transformations and Spark actions, were covered using Scala and Python APIs. All the important features of the Spark programming model were covered with the help of real-world use cases. This chapter also discussed the library stack that comes with Spark and what each one is doing. In summary, Spark comes with a very user-friendly programming model and in turn provides a very powerful data processing toolset.

The next chapter will discuss the Dataset API and the DataFrame API. The Dataset API is going to be the new way of programming with Spark, while the DataFrame API deals with more structured data. Spark SQL is also introduced to manipulate structured data and show how that can be intermixed with any Spark data processing application.

3
Spark SQL

Most businesses deal with quite a lot of structured data all the time. Even if there are too many ways to deal with unstructured data, many application use cases still have to have structured data. What is the major difference between processing structured data and unstructured data? If the data source is structured, and if the data processing engine knows the data structure a priori, the data processing engine can do lots of optimizations while processing the data, or even beforehand. This is very crucial when the data processing volume is huge and the turn around time is very critical.

Proliferation of enterprise data mandated the need to empower the end users to query and process the data in simple and easy to use application user interfaces. The RDBMS vendors united and the **structured query language** (**SQL**) came about as a solution for this. Over the last couple of decades, everyone who deals with data became familiar with SQL if not power users.

The large scale Internet applications in the social networking and microblogging spaces, to name a few, produced data beyond the consumption of many traditional data processing tools. When dealing with such a sea of data, picking and choosing the right piece of data from it became even more important. Spark was a highly prevalent data processing platform and its RDD-based programming model reduced the data processing effort as compared to the Hadoop MapReduce data processing framework. But, the initial versions of Spark's RDD-based programming model remained elusive on making end users, such as data scientists, data analysts, and business analysts from using Spark. The main reason why they could not make use of RDD based Spark programming model is because it requires some amount of functional programming. The solution to this problem is Spark SQL. Spark SQL is a library built on top of Spark. It exposes SQL interface and DataFrame API. DataFrame API supports programming languages Scala, Java, Python, and R.

If the structure of the data is known in advance, if the data fits into the model of rows and columns, it doesn't matter from where the data is coming and Spark SQL can use all of it together and process it as if all the data is coming from a single source. Moreover, the

querying dialect is the ubiquitous SQL.

We will cover the following topics in this chapter:

- Structure of data
- Spark SQL
- Aggregations
- Multi-datasource joins
- Dataset
- Data catalog

Understanding the structure of data

The structure of the data that is being discussed here needs some more elucidation. What do we mean by the structure of the data? The data stored in RDBMS has a way of storing the data in rows/columns or records/fields. Every field has a data type and every record is a collection of fields of the same or different data types. In the early days of RDBMS, the data types of the fields were scalar and in the recent versions, it expanded to include collection data types or composite data types as well. So, whether the record contains scalar data types or composite data types, the important point to note here is that there is a structure to the underlying data. Many of the data processing paradigms have adopted the concept of mirroring the underlying data structure persisted in the RDBMS or other stores in memory to make the data processing easy.

In other words, if the data in an RDBMS table is being processed by a data processing application, if the same table-like data structure is available in memory to the programs, for the end users and programmers it is easy to model the applications and query the data. For example, suppose there is a set of comma-separated data items with a fixed number of values in each row having a specific data type for the values coming in the specific position in all the rows. This is a structured data file. It is a data table and is very similar to an RDBMS table.

In programming languages such as R, there is a data frame abstraction used to store data tables in memory. The Python data analysis library, named Pandas, also has a similar data frame concept. Once that data structure is available in memory, the programs can extract the data and slice and dice it as per the need. The same data table concept is extended to Spark, known as DataFrame, built on top of RDD, and there is a very comprehensive API known as DataFrame API in Spark SQL to process the data in the DataFrame. A SQL-like query language is also developed on top of the DataFrame abstraction, catering to the needs of the end users to query and process the underlying structured data. In summary,

DataFrame is a distributed data table organized in rows and columns and having names for each column.

The Spark SQL library built on top of Spark is developed based on the research paper titled *"Spark SQL: Relational Data Processing in Spark"*. It talks about four goals for Spark SQL and they are reproduced verbatim as follows:

- Support relational processing both within Spark programs (on native RDDs) and on external data sources using a programmer-friendly API
- Provide high performance using established DBMS techniques
- Easily support new data sources, including semi-structured data and external databases amenable to query federation
- Enable extension with advanced analytics algorithms such as graph processing and machine learning

DataFrame holds structured data, and it is distributed. It allows selection, filtering, and aggregation of data. Sounding very similar to RDD? The key difference between RDD and DataFrame is that DataFrame stores much more information about the structure of the data, such as the data types and names of the columns, than RDD. This allows the DataFrame to optimize the processing much more effectively than Spark transformations and Spark actions doing processing on RDD. The other most important aspect to mention here is that all the supported programming languages of Spark can be used to develop applications using the DataFrame API of Spark SQL. For all practical purposes, Spark SQL is a distributed SQL engine.

> Those who have worked earlier to Spark 1.3 must be familiar with SchemaRDD, and the concept of DataFrame is exactly built on top of SchemaRDD with API-level compatibility.

Why Spark SQL?

There is no doubt that SQL is the lingua franca for doing data analysis and Spark SQL is the answer from the Spark family of toolsets to do data analysis. So, what does it provide? It provides the ability to run SQL on top of Spark. Whether the data is coming from CSV, Avro, Parquet, Hive, NoSQL data stores such as Cassandra, or even RDBMS, Spark SQL can be used to analyze data and mix in with Spark programs. Many of the data sources mentioned here are supported intrinsically by Spark SQL and many others are supported by external packages. The most important aspect to highlight here is the ability of Spark SQL to deal with data from a very wide variety of data sources. Once it is available as a

DataFrame in Spark, Spark SQL can process data in a completely distributed way, combining the DataFrames coming from various data sources to process and query as if the entire dataset were coming from a single source.

In the previous chapter, the RDD was discussed in detail and introduced as the Spark programming model. Is the DataFrames API and the usage of SQL dialects in Spark SQL replacing RDD-based programming model? Definitely not! The RDD-based programming model is the generic and the basic data processing model in Spark. RDD-based programming requires the use of real programming techniques. The Spark transformations and Spark actions use a lot of functional programming constructs. Even though the amount of code that is required to be written in the RDD-based programming model is less compared to Hadoop MapReduce or any other paradigm, there is still a need to write some amount of functional code. This is a barrier for many data scientists, data analysts, and business analysts, who may perform major exploratory kinds of data analysis or do some prototyping with the data. Spark SQL completely removes those constraints. Simple and easy-to-use **domain specific language** (**DSL**) based methods to read and write data from data sources, SQL-like language to select, filter, and aggregate, and the capability to read data from a wide variety of data sources, make it easy for anybody who knows the data structure to use it.

> What is the best use case to use RDD and which is the best use case to use Spark SQL? The answer is very simple. If the data is structured, if it can be arranged in tables, and if each column can be given a name, then use Spark SQL. This doesn't mean that the RDD and DataFrame are two disparate entities. They interoperate very well. Conversions from RDD to DataFrame and vice versa are very much possible. Many of the Spark transformations and Spark actions that are typically applied on RDDs can also be applied on DataFrames.

Typically, during the application design phase, business analysts generally do lots of analysis with the application data using SQL, and that is fed to the application requirements and testing artifacts. While designing big data applications, the same thing is needed, and in such situations, apart from business analysts, data scientists will also be there in the team. In a Hadoop-based ecosystem, Hive is used extensively for data analysis with big data. Now Spark SQL brings that capability to any platform with support for a whole lot of data sources. If there is a standalone Spark installation on commodity hardware, lots of these kinds of activities can be done to analyze the data. A basic Spark installation deployed in standalone mode on commodity hardware is enough to play around with a whole lot of data.

The SQL-on-Hadoop strategy has introduced many applications, such as Hive and Impala to name a few, providing a SQL-like interface to the underlying big data stored in the

Hadoop Distributed File System (**HDFS**). Where does Spark SQL fit in that space? Before jumping into that, it is a good idea to touch upon Hive and Impala. Hive is a MapReduce-based data warehousing technology and, because of the use of MapReduce to process queries, Hive queries require lots of I/O operations before completing a query. Impala came up with a brilliant solution by doing the in-memory processing while making use of the Hive meta store that describes the data. Spark SQL uses SQLContext to do all the operations with data. But it can also use HiveContext, which is much more feature rich and advanced than SQLContext. HiveContext can do all that SQLContext can do and, on top of that, it can read from Hive meta store and tables, and can access Hive user-defined functions as well. The only requirement to use HiveContext is obviously that there should be an already existing Hive setup readily available. In this way, Spark SQL can easily co-exist with Hive.

> From Spark 2.0 onwards, SparkSession is the new starting point for Spark SQL-based applications, which are a combination of SQLContext and HiveContext while supporting backward compatibility with SQLContext and HiveContext.

Spark SQL can process the data from Hive tables faster than Hive using its Hive Query Language. Another very interesting feature of Spark SQL is that it can read data from different versions of Hive, which is a great feature enabling data source consolidation for data processing.

> The library that exposes Spark SQL and DataFrame API provides interfaces that can be accessed through JDBC/ODBC. This opens up a whole new world of data analysis. For example, a **business intelligence** (**BI**) tool that connects to data sources using JDBC/ODBC can use a whole lot of data sources supported by Spark SQL. Moreover, the BI tools can push down the processor intensive join aggregation operations to a huge cluster of worker nodes in the Spark infrastructure.

Anatomy of Spark SQL

Interaction with Spark SQL library is done mainly through two methods. One is through SQL-like queries and the other is through DataFrame API. Before getting into the details of how DataFrame-based programs work, it is a good idea to see how the RDD-based programs work.

The Spark transformations and Spark actions are converted into Java functions and they act on top of RDDs, which are nothing but Java objects acting upon data. Since RDD is a pure Java object, there is no way, at compile time or at run time, to know about what data is going to process. There is no metadata available to the execution engine beforehand to

optimize the Spark transformations or Spark actions. There are no multiple execution paths or query plans available in advance to process that data and so, evaluation of the efficacy of various paths of execution is not available.

Here, there is no optimized query plan executed because there is no schema associated, with data. In the case of DataFrame, the structure is well-known in advance. Because of this the queries can be optimized and data cache can be built beforehand.

The following *Figure 1* gives an idea about the same:
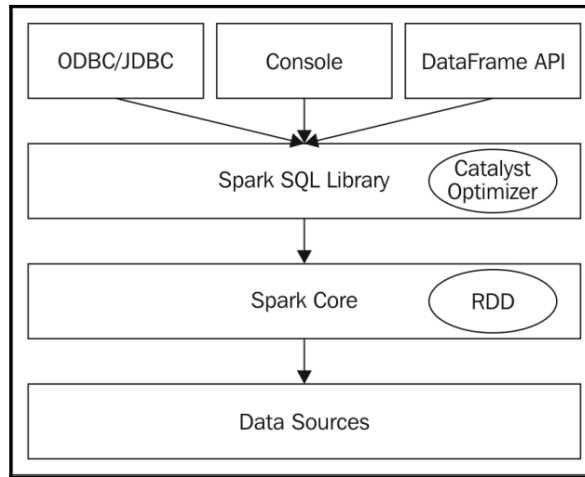
Figure 1

The SQL-like queries and DataFrame API calls made against DataFrame are converted to language-neutral expressions. The language-neutral expression corresponding to a SQL query or DataFrame API is called an unresolved logical plan.

The unresolved logical plan is converted to a logical plan by doing validations on column names from the metadata of the DataFrame. The logical plan is further optimized by applying standard rules such as simplification of expressions, evaluations of expressions, and other optimization rules, to form an optimized logical plan. The optimized logical plan is converted to multiple physical plans. The physical plans are created by using Spark-specific operators in the logical plan. The best physical plan is chosen and the resultant queries are pushed down to RDDs to act on the data. Because the SQL queries and DataFrame API calls are converted to language-neutral query expressions, the performance of these queries is consistent across all the supported languages. That is the same reason why the DataFrame API is supported by all the Spark supported languages such as Scala, Java, Python, and R. In the future, there is a good chance that many more languages are going to be supporting DataFrame API and Spark SQL because of this reason.

The query planning and optimizations of Spark SQL are worth mentioning here too. Any query operation done on a DataFrame through SQL queries or through DataFrame API is highly optimized before the corresponding operations are physically applied on the underlying base RDD. There are many processes in between before the real action happening on the RDD.

Figure 2 gives some idea about the whole query optimization process:
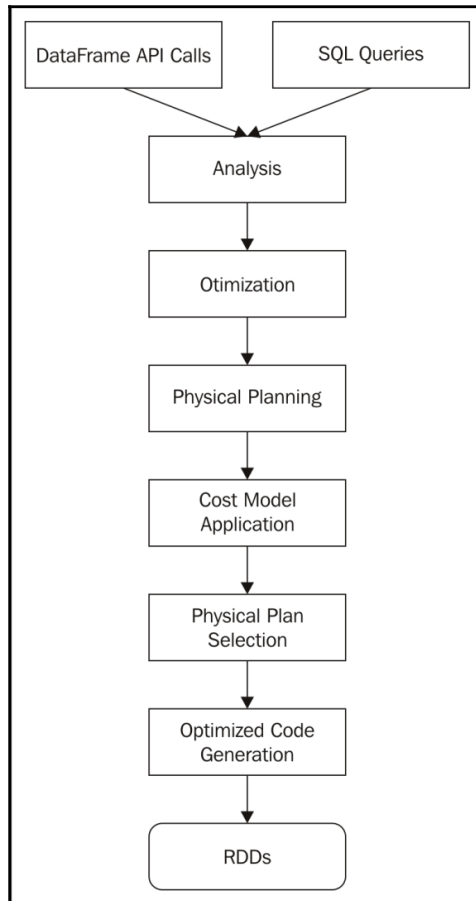
Figure 2

Two types of queries can be called against a DataFrame. They are SQL queries or DataFrame API calls. They go through a proper analysis to come up with a logical query plan of execution. Then, optimizations are applied on the logical query plans to arrive at an optimized logical query plan. From the final optimized logical query plan, one or more physical query plans are made. For each of the physical query plans, cost models are worked out, and based on the optimal cost, an appropriate physical query plan is selected, and highly optimized code is generated and run against the RDDs. This is the reason behind the consistent performance of queries of any type on DataFrame. This is the same reason why the DataFrame API calls from all these different languages, Scala, Java, Python, and R, give consistent performance.

Let's revisit the bigger picture once again, as given in *Figure 3*, to set the context and see what is being discussed here before getting into and taking up the use cases:
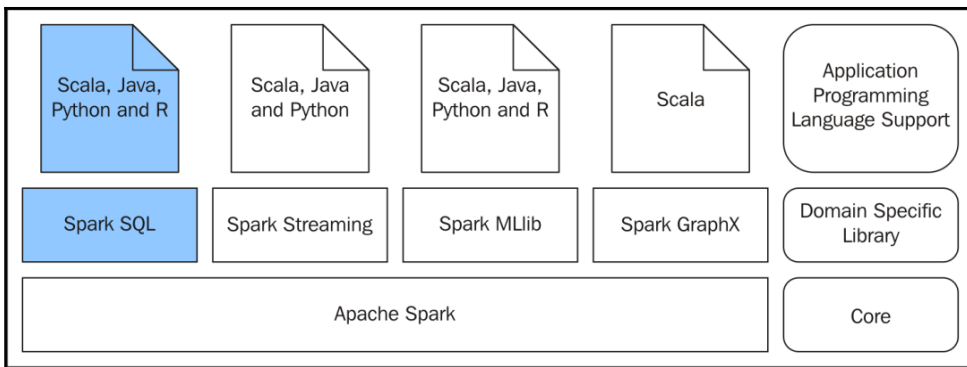
Figure 3

The use cases that are going to be discussed here will demonstrate the ability to mix SQL queries with Spark programs. Multiple data sources will be chosen, data will be read from those sources using DataFrame, and uniform data access will be demonstrated. The programming languages used to demonstrate are still Scala and Python. The usage of R to manipulate DataFrames is on the agenda of the book and a whole chapter is dedicated to the same.

DataFrame programming

The use cases selected for elucidating the Spark SQL way of programming with DataFrame are given as follows:

- The transaction records come as comma-separated values.
- Filter out only the good transaction records from the list. The account number should start with SB and the transaction amount should be greater than zero.
- Find all the high-value transaction records with a transaction amount greater than 1000.
- Find all the transaction records where the account number is bad.
- Find all the transaction records where the transaction amount is less than or equal to zero.
- Find a combined list of all the bad transaction records.
- Find the total of all the transaction amounts.
- Find the maximum of all the transaction amounts.
- Find the minimum of all the transaction amounts.
- Find all the good account numbers.

This is exactly the same set of use cases that were used in the previous chapter as well, but here the programming model is totally different. Using this set of use cases, two types of programming models are demonstrated here. One is using the SQL queries and the other is using DataFrame APIs.

Programming with SQL

At the Scala REPL prompt, try the following statements:

```scala
scala> // Define the case classes for using in conjunction with DataFrames
scala> case class Trans(accNo: String, tranAmount: Double)
defined class Trans
scala> // Functions to convert the sequence of strings to objects defined
by the case classes
scala> def toTrans =  (trans: Seq[String]) => Trans(trans(0),
trans(1).trim.toDouble)
toTrans: Seq[String] => Trans
scala> // Creation of the list from where the RDD is going to be created
scala> val acTransList = Array("SB10001,1000", "SB10002,1200",
"SB10003,8000", "SB10004,400", "SB10005,300", "SB10006,10000",
"SB10007,500", "SB10008,56", "SB10009,30","SB10010,7000", "CR10001,7000",
"SB10002,-10")
```

```
acTransList: Array[String] = Array(SB10001,1000, SB10002,1200,
SB10003,8000, SB10004,400, SB10005,300, SB10006,10000, SB10007,500,
SB10008,56, SB10009,30, SB10010,7000, CR10001,7000, SB10002,-10)
scala> // Create the RDD
scala> val acTransRDD =
sc.parallelize(acTransList).map(_.split(",")).map(toTrans(_))
acTransRDD: org.apache.spark.rdd.RDD[Trans] = MapPartitionsRDD[2] at map at
<console>:30
scala> // Convert RDD to DataFrame
scala> val acTransDF = spark.createDataFrame(acTransRDD)
acTransDF: org.apache.spark.sql.DataFrame = [accNo: string, tranAmount:
double]
scala> // Register temporary view in the DataFrame for using it in SQL
scala> acTransDF.createOrReplaceTempView("trans")
scala> // Print the structure of the DataFrame
scala> acTransDF.printSchema
root
 |-- accNo: string (nullable = true)
 |-- tranAmount: double (nullable = false)
scala> // Show the first few records of the DataFrame
scala> acTransDF.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
scala> // Use SQL to create another DataFrame containing the good
transaction records
scala> val goodTransRecords = spark.sql("SELECT accNo, tranAmount FROM
trans WHERE accNo like 'SB%' AND tranAmount > 0")
goodTransRecords: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
scala> // Register temporary view in the DataFrame for using it in SQL
scala> goodTransRecords.createOrReplaceTempView("goodtrans")
scala> // Show the first few records of the DataFrame
scala> goodTransRecords.show
+-------+----------+
```

```
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
+-------+----------+
scala> // Use SQL to create another DataFrame containing the high value
transaction records
scala> val highValueTransRecords = spark.sql("SELECT accNo, tranAmount FROM
goodtrans WHERE tranAmount > 1000")
highValueTransRecords: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> highValueTransRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10006|   10000.0|
|SB10010|    7000.0|
+-------+----------+
scala> // Use SQL to create another DataFrame containing the bad account
records
scala> val badAccountRecords = spark.sql("SELECT accNo, tranAmount FROM
trans WHERE accNo NOT like 'SB%'")
badAccountRecords: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> badAccountRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
+-------+----------+
scala> // Use SQL to create another DataFrame containing the bad amount
records
scala> val badAmountRecords = spark.sql("SELECT accNo, tranAmount FROM
trans WHERE tranAmount < 0")
badAmountRecords: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
```

```
scala> // Show the first few records of the DataFrame
scala> badAmountRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     -10.0|
+-------+----------+
scala> // Do the union of two DataFrames and create another DataFrame
scala> val badTransRecords = badAccountRecords.union(badAmountRecords)
badTransRecords: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[accNo: string, tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> badTransRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
scala> // Calculate the sum
scala> val sumAmount = spark.sql("SELECT sum(tranAmount) as sum FROM
goodtrans")
sumAmount: org.apache.spark.sql.DataFrame = [sum: double]
scala> // Show the first few records of the DataFrame
scala> sumAmount.show
+-------+
|    sum|
+-------+
|28486.0|
+-------+
scala> // Calculate the maximum
scala> val maxAmount = spark.sql("SELECT max(tranAmount) as max FROM
goodtrans")
maxAmount: org.apache.spark.sql.DataFrame = [max: double]
scala> // Show the first few records of the DataFrame
scala> maxAmount.show
+-------+
|    max|
+-------+
|10000.0|
+-------+
scala> // Calculate the minimum
scala> val minAmount = spark.sql("SELECT min(tranAmount) as min FROM
goodtrans")
minAmount: org.apache.spark.sql.DataFrame = [min: double]
scala> // Show the first few records of the DataFrame
scala> minAmount.show
+----+
```

```
| min|
+----+
|30.0|
+----+
scala> // Use SQL to create another DataFrame containing the good account
numbers
scala> val goodAccNos = spark.sql("SELECT DISTINCT accNo FROM trans WHERE
accNo like 'SB%' ORDER BY accNo")
goodAccNos: org.apache.spark.sql.DataFrame = [accNo: string]
scala> // Show the first few records of the DataFrame
scala> goodAccNos.show
+-------+
|  accNo|
+-------+
|SB10001|
|SB10002|
|SB10003|
|SB10004|
|SB10005|
|SB10006|
|SB10007|
|SB10008|
|SB10009|
|SB10010|
+-------+
scala> // Calculate the aggregates using mixing of DataFrame and RDD like
operations
scala> val sumAmountByMixing = goodTransRecords.map(trans =>
trans.getAs[Double]("tranAmount")).reduce(_ + _)
sumAmountByMixing: Double = 28486.0
scala> val maxAmountByMixing = goodTransRecords.map(trans =>
trans.getAs[Double]("tranAmount")).reduce((a, b) => if (a > b) a else b)
maxAmountByMixing: Double = 10000.0
scala> val minAmountByMixing = goodTransRecords.map(trans =>
trans.getAs[Double]("tranAmount")).reduce((a, b) => if (a < b) a else b)
minAmountByMixing: Double = 30.0
```

The retail banking transaction records come with account number, transaction amount and are processed using SparkSQL to get the desired results of the use cases. Here is the summary of what the preceding script did:

- A Scala case class is defined to describe the structure of the transaction record to be fed into the DataFrame.
- An array is defined with the necessary transaction records.

- An RDD is made from the array, split the comma-separated values, mapped it to create objects using the Scala case class that was defined as the first step in the scripts, and the RDD is converted to a DataFrame. This is one use case of interoperability between RDD and DataFrame.
- A table is registered with the DataFrame with a name. This registered name of the table can be used in SQL statements.
- Then, all the other activities are just issuing SQL statements using the `spark.sql` method. Here the object spark is of type the SparkSession.
- The result of all these SQL statements is stored as DataFrames and, just like the RDD's `collect` action, DataFrame's show method is used to extract the values to the Spark driver program.
- The aggregate value calculations are done in two different ways. One is in the SQL statement way, which is the easiest way. The other is using the regular RDD-style Spark transformations and Spark actions. This is to show that even DataFrame can be operated like an RDD, and Spark transformations and Spark actions can be applied on top of DataFrame.
- At times, it is easy to do some data manipulation activities through the functional style operations using functions. So, there is a flexibility here to mix SQL, RDD, and DataFrame to have a very convenient programming model to process data.
- The DataFrame contents are displayed in table format using the `show` method of the DataFrame.
- A detailed view of the structure of the DataFrame is displayed using the `printSchema` method. This is akin to the `describe` command of the database tables.

At the Python REPL prompt, try the following statements:

```
>>> from pyspark.sql import Row
>>> # Creation of the list from where the RDD is going to be created
>>> acTransList = ["SB10001,1000", "SB10002,1200", "SB10003,8000",
"SB10004,400", "SB10005,300", "SB10006,10000", "SB10007,500", "SB10008,56",
"SB10009,30","SB10010,7000", "CR10001,7000", "SB10002,-10"]
>>> # Create the DataFrame
>>> acTransDF = sc.parallelize(acTransList).map(lambda trans:
trans.split(",")).map(lambda p: Row(accNo=p[0],
tranAmount=float(p[1]))).toDF()
>>> # Register temporary view in the DataFrame for using it in SQL
>>> acTransDF.createOrReplaceTempView("trans")
>>> # Print the structure of the DataFrame
>>> acTransDF.printSchema()
root
 |-- accNo: string (nullable = true)
```

```
  |-- tranAmount: double (nullable = true)
>>> # Show the first few records of the DataFrame
>>> acTransDF.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
>>> # Use SQL to create another DataFrame containing the good transaction
records
>>> goodTransRecords = spark.sql("SELECT accNo, tranAmount FROM trans WHERE
accNo like 'SB%' AND tranAmount > 0")
>>> # Register temporary table in the DataFrame for using it in SQL
>>> goodTransRecords.createOrReplaceTempView("goodtrans")
>>> # Show the first few records of the DataFrame
>>> goodTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
+-------+----------+
>>> # Use SQL to create another DataFrame containing the high value
transaction records
>>> highValueTransRecords = spark.sql("SELECT accNo, tranAmount FROM
goodtrans WHERE tranAmount > 1000")
>>> # Show the first few records of the DataFrame
>>> highValueTransRecords.show()
+-------+----------+
```

```
|  accNo|tranAmount|
+-------+----------+
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10006|   10000.0|
|SB10010|    7000.0|
+-------+----------+
>>> # Use SQL to create another DataFrame containing the bad account
records
>>> badAccountRecords = spark.sql("SELECT accNo, tranAmount FROM trans
WHERE accNo NOT like 'SB%'")
>>> # Show the first few records of the DataFrame
>>> badAccountRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
+-------+----------+
>>> # Use SQL to create another DataFrame containing the bad amount records
>>> badAmountRecords = spark.sql("SELECT accNo, tranAmount FROM trans WHERE
tranAmount < 0")
>>> # Show the first few records of the DataFrame
>>> badAmountRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     -10.0|
+-------+----------+
>>> # Do the union of two DataFrames and create another DataFrame
>>> badTransRecords = badAccountRecords.union(badAmountRecords)
>>> # Show the first few records of the DataFrame
>>> badTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
>>> # Calculate the sum
>>> sumAmount = spark.sql("SELECT sum(tranAmount)as sum FROM goodtrans")
>>> # Show the first few records of the DataFrame
>>> sumAmount.show()
+-------+
|    sum|
+-------+
|28486.0|
+-------+
>>> # Calculate the maximum
```

```
>>> maxAmount = spark.sql("SELECT max(tranAmount) as max FROM goodtrans")
>>> # Show the first few records of the DataFrame
>>> maxAmount.show()
+-------+
|    max|
+-------+
|10000.0|
+-------+
>>> # Calculate the minimum
>>> minAmount = spark.sql("SELECT min(tranAmount)as min FROM goodtrans")
>>> # Show the first few records of the DataFrame
>>> minAmount.show()
+----+
| min|
+----+
|30.0|
+----+
>>> # Use SQL to create another DataFrame containing the good account
numbers
>>> goodAccNos = spark.sql("SELECT DISTINCT accNo FROM trans WHERE accNo
like 'SB%' ORDER BY accNo")
>>> # Show the first few records of the DataFrame
>>> goodAccNos.show()
+-------+
|  accNo|
+-------+
|SB10001|
|SB10002|
|SB10003|
|SB10004|
|SB10005|
|SB10006|
|SB10007|
|SB10008|
|SB10009|
|SB10010|
+-------+
>>> # Calculate the sum using mixing of DataFrame and RDD like operations
>>> sumAmountByMixing = goodTransRecords.rdd.map(lambda trans:
trans.tranAmount).reduce(lambda a,b : a+b)
>>> sumAmountByMixing
28486.0
>>> # Calculate the maximum using mixing of DataFrame and RDD like
operations
>>> maxAmountByMixing = goodTransRecords.rdd.map(lambda trans:
trans.tranAmount).reduce(lambda a,b : a if a > b else b)
>>> maxAmountByMixing
10000.0
```

```
>>> # Calculate the minimum using mixing of DataFrame and RDD like
operations
>>> minAmountByMixing = goodTransRecords.rdd.map(lambda trans:
trans.tranAmount).reduce(lambda a,b : a if a < b else b)
>>> minAmountByMixing
30.0
```

In the preceding Python code snippet, except for a few language-specific constructs such as importing libraries and the definition of lambda functions, the style of programming is almost the same, most of the time, as the Scala code. This is the advantage of Spark's uniform programming model. As discussed earlier, when business analysts or data analysts provide the SQL for data access, it is very easy to integrate that along with the data processing code in Spark. This uniform programming style of coding is very useful for organizations to use the language of their choice for developing data processing applications in Spark.

> On DataFrames, if applicable Spark transformations are applied, then a Dataset is returned instead of a DataFrame. The concept of Dataset is introduced toward the end of this chapter. There is a very strong relationship between DataFrame and Dataset, and that is explained in the section covering Datasets. While developing applications, caution must be used in this kind of situation. For example, in the preceding code snippets, if the following transformation is tried in Scala REPL, it will return a dataset: `val amount = goodTransRecords.map(trans => trans.getAs[Double]("tranAmount"))amount: org.apache.spark.sql.Dataset[Double] = [value: double]`

Programming with DataFrame API

In this section, the code snippets will be run in the appropriate language REPLs as a continuation of the previous section so that the setup of the data and other initializations are not repeated. Like the preceding code snippets, initially, some DataFrame-specific basic commands are given. These are used regularly to see the contents and for doing some sanity tests on the DataFrame and its contents. These are commands that are typically used in the exploratory stage of the data analysis, quite often to get more insight into the structure and contents of the underlying data.

At the Scala REPL prompt, try the following statements:

```
scala> acTransDF.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
scala> // Create the DataFrame using API for the good transaction records
scala> val goodTransRecords = acTransDF.filter("accNo like
'SB%'").filter("tranAmount > 0")
goodTransRecords: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[accNo: string, tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> goodTransRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
+-------+----------+
scala> // Create the DataFrame using API for the high value transaction
records
scala> val highValueTransRecords = goodTransRecords.filter("tranAmount >
1000")
highValueTransRecords:
org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [accNo: string,
tranAmount: double]
scala> // Show the first few records of the DataFrame
```

```
scala> highValueTransRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10006|   10000.0|
|SB10010|    7000.0|
+-------+----------+
scala> // Create the DataFrame using API for the bad account records
scala> val badAccountRecords = acTransDF.filter("accNo NOT like 'SB%'")
badAccountRecords: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[accNo: string, tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> badAccountRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
+-------+----------+
scala> // Create the DataFrame using API for the bad amount records
scala> val badAmountRecords = acTransDF.filter("tranAmount < 0")
badAmountRecords: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[accNo: string, tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> badAmountRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     -10.0|
+-------+----------+
scala> // Do the union of two DataFrames
scala> val badTransRecords = badAccountRecords.union(badAmountRecords)
badTransRecords: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[accNo: string, tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> badTransRecords.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
scala> // Calculate the aggregates in one shot
scala> val aggregates = goodTransRecords.agg(sum("tranAmount"),
max("tranAmount"), min("tranAmount"))
aggregates: org.apache.spark.sql.DataFrame = [sum(tranAmount): double,
max(tranAmount): double ... 1 more field]
```

```
scala> // Show the first few records of the DataFrame
scala> aggregates.show
+---------------+---------------+---------------+
|sum(tranAmount)|max(tranAmount)|min(tranAmount)|
+---------------+---------------+---------------+
|        28486.0|        10000.0|           30.0|
+---------------+---------------+---------------+
scala> // Use DataFrame using API for creating the good account numbers
scala> val goodAccNos = acTransDF.filter("accNo like
'SB%'").select("accNo").distinct().orderBy("accNo")
goodAccNos: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
[accNo: string]
scala> // Show the first few records of the DataFrame
scala> goodAccNos.show
+-------+
|  accNo|
+-------+
|SB10001|
|SB10002|
|SB10003|
|SB10004|
|SB10005|
|SB10006|
|SB10007|
|SB10008|
|SB10009|
|SB10010|
+-------+
scala> // Persist the data of the DataFrame into a Parquet file
scala> acTransDF.write.parquet("scala.trans.parquet")
scala> // Read the data into a DataFrame from the Parquet file
scala> val acTransDFfromParquet = spark.read.parquet("scala.trans.parquet")
acTransDFfromParquet: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
scala> // Show the first few records of the DataFrame
scala> acTransDFfromParquet.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10008|      56.0|
|SB10009|      30.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
|SB10001|    1000.0|
```

```
|SB10004|     400.0|
|SB10007|     500.0|
|SB10010|    7000.0|
+-------+----------+
```

Here is the summary of what the preceding script did from a DataFrame API perspective:

- The DataFrame containing the superset of data used in the preceding section is used here.
- Filtering of the records is demonstrated next. Here, the most important aspect to notice is that the filter predicate is to be given exactly like the predicates in the SQL statements. Filters can be chained.
- The aggregation methods are calculated in one go as three columns in the resultant DataFrame.
- The final statements in this set are doing the selection, filtering, choosing distinct records, and ordering in one single chained statement.
- Finally, the transaction records are persisted in Parquet format, read from the Parquet store and create a DataFrame. More details on the persistence formats is coming in the following section.
- In this code snippet, the Parquet format data is stored in the current directory from where the corresponding REPL is invoked. When it is run as a Spark program, the directory again will be the current directory from where the Spark submit is invoked.

At the Python REPL prompt, try the following statements:

```
>>> acTransDF.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
>>> # Print the structure of the DataFrame
```

```
>>> acTransDF.printSchema()
root
 |-- accNo: string (nullable = true)
 |-- tranAmount: double (nullable = true)
>>> # Create the DataFrame using API for the good transaction records
>>> goodTransRecords = acTransDF.filter("accNo like
'SB%'").filter("tranAmount > 0")
>>> # Show the first few records of the DataFrame
>>> goodTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
+-------+----------+
>>> # Create the DataFrame using API for the high value transaction records
>>> highValueTransRecords = goodTransRecords.filter("tranAmount > 1000")
>>> # Show the first few records of the DataFrame
>>> highValueTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10006|   10000.0|
|SB10010|    7000.0|
+-------+----------+
>>> # Create the DataFrame using API for the bad account records
>>> badAccountRecords = acTransDF.filter("accNo NOT like 'SB%'")
>>> # Show the first few records of the DataFrame
>>> badAccountRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
+-------+----------+
>>> # Create the DataFrame using API for the bad amount records
>>> badAmountRecords = acTransDF.filter("tranAmount < 0")
>>> # Show the first few records of the DataFrame
>>> badAmountRecords.show()
```

```
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     -10.0|
+-------+----------+
>>> # Do the union of two DataFrames and create another DataFrame
>>> badTransRecords = badAccountRecords.union(badAmountRecords)
>>> # Show the first few records of the DataFrame
>>> badTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
>>> # Calculate the sum
>>> sumAmount = goodTransRecords.agg({"tranAmount": "sum"})
>>> # Show the first few records of the DataFrame
>>> sumAmount.show()
+--------------+
|sum(tranAmount)|
+--------------+
|       28486.0|
+--------------+
>>> # Calculate the maximum
>>> maxAmount = goodTransRecords.agg({"tranAmount": "max"})
>>> # Show the first few records of the DataFrame
>>> maxAmount.show()
+--------------+
|max(tranAmount)|
+--------------+
|       10000.0|
+--------------+
>>> # Calculate the minimum
>>> minAmount = goodTransRecords.agg({"tranAmount": "min"})
>>> # Show the first few records of the DataFrame
>>> minAmount.show()
+--------------+
|min(tranAmount)|
+--------------+
|          30.0|
+--------------+
>>> # Create the DataFrame using API for the good account numbers
>>> goodAccNos = acTransDF.filter("accNo like
'SB%'").select("accNo").distinct().orderBy("accNo")
>>> # Show the first few records of the DataFrame
>>> goodAccNos.show()
+-------+
```

```
|  accNo|
+-------+
|SB10001|
|SB10002|
|SB10003|
|SB10004|
|SB10005|
|SB10006|
|SB10007|
|SB10008|
|SB10009|
|SB10010|
+-------+
>>> # Persist the data of the DataFrame into a Parquet file
>>> acTransDF.write.parquet("python.trans.parquet")
>>> # Read the data into a DataFrame from the Parquet file
>>> acTransDFfromParquet = spark.read.parquet("python.trans.parquet")
>>> # Show the first few records of the DataFrame
>>> acTransDFfromParquet.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10008|      56.0|
|SB10009|      30.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
|SB10001|    1000.0|
|SB10004|     400.0|
|SB10007|     500.0|
|SB10010|    7000.0|
+-------+----------+
```

In the preceding Python code snippet, except for a very few variations in the aggregation calculations, the programming constructs are almost similar to its Scala counterpart.

The last few statements of the preceding Scala and Python sections are about the persisting of the DataFrame contents into the media. The writing and reading operations are very much required in any kind of data processing operations, but most of the tools don't have a uniform way of writing and reading. Spark SQL is different. The DataFrame API comes with a rich set of persistence mechanisms. It is very easy to write contents of a DataFrame into many supported persistence stores. All these writing and reading operations have very simple DSL style interfaces. Here are some of the built-in formats in which DataFrames can be written to and read from.

Apart from these, there are so many other external data sources supported through third-party packages:

- JSON
- Parquet
- Hive
- MySQL
- PostgreSQL
- HDFS
- Plain Text
- Amazon S3
- ORC
- JDBC

The write and read of DataFrame into and from Parquet has been demonstrated in the preceding code snippets. All the preceding inherently supported data stores have very simple DSL style syntax for persistence and reading back, which makes the programming style uniform once again. The DataFrame API reference is a great source to know about the details of dealing with each of these data stores.

The sample code in this chapter persists data in Parquet and JSON formats. The data store location names chosen are `python.trans.parquet`, `scala.trans.parquet`, and so on. This is just to give an indication of which programming language is used and which is the format of the data. This is not a proper convention but a convenience. When one run of the program is completed, these directories would have been created. Next time the same program is run, it will try to create the same and will result in an error. The workaround is to remove the directories manually, before the subsequent runs, and proceed. Proper error handling mechanisms and other nuances of fine programming are going to dilute the focus and hence are deliberately left out of this book.

Understanding Aggregations in Spark SQL

In SQL, aggregation of data is very flexible. The same thing is true in Spark SQL too. Instead of running SQL statements on a single data source located in a single machine, here Spark SQL can do the same on distributed data sources. In the previous chapter, a MapReduce use case was discussed to do data aggregation and the same is being used here to demonstrate the aggregation capabilities of Spark SQL. In this section also, the use cases are approached in the SQL query way as well as in the DataFrame API way.

The use cases selected for elucidating the MapReduce kind of data processing here are given as follows:

- The retail banking transaction records come with account number and transaction amount in comma-separated strings
- Find an account level summary of all the transactions to get the account balance

At the Scala REPL prompt, try the following statements:

```scala
scala> // Define the case classes for using in conjunction with DataFrames
scala> case class Trans(accNo: String, tranAmount: Double)
defined class Trans
scala> // Functions to convert the sequence of strings to objects defined
by the case classes
scala> def toTrans =  (trans: Seq[String]) => Trans(trans(0),
trans(1).trim.toDouble)
toTrans: Seq[String] => Trans
scala> // Creation of the list from where the RDD is going to be created
scala> val acTransList = Array("SB10001,1000",
"SB10002,1200","SB10001,8000", "SB10002,400", "SB10003,300",
"SB10001,10000","SB10004,500","SB10005,56",
"SB10003,30","SB10002,7000","SB10001,-100", "SB10002,-10")
acTransList: Array[String] = Array(SB10001,1000, SB10002,1200,
SB10001,8000, SB10002,400, SB10003,300, SB10001,10000, SB10004,500,
SB10005,56, SB10003,30, SB10002,7000, SB10001,-100, SB10002,-10)
scala> // Create the DataFrame
scala> val acTransDF =
sc.parallelize(acTransList).map(_.split(",")).map(toTrans(_)).toDF()
acTransDF: org.apache.spark.sql.DataFrame = [accNo: string, tranAmount:
double]
scala> // Show the first few records of the DataFrame
scala> acTransDF.show
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10001|    8000.0|
|SB10002|     400.0|
|SB10003|     300.0|
|SB10001|   10000.0|
|SB10004|     500.0|
|SB10005|      56.0|
|SB10003|      30.0|
|SB10002|    7000.0|
|SB10001|    -100.0|
|SB10002|     -10.0|
```

```
+-------+----------+
scala> // Register temporary view in the DataFrame for using it in SQL
scala> acTransDF.createOrReplaceTempView("trans")
scala> // Use SQL to create another DataFrame containing the account
summary records
scala> val acSummary = spark.sql("SELECT accNo, sum(tranAmount) as
TransTotal FROM trans GROUP BY accNo")
acSummary: org.apache.spark.sql.DataFrame = [accNo: string, TransTotal:
double]
scala> // Show the first few records of the DataFrame
scala> acSummary.show
+-------+----------+
|  accNo|TransTotal|
+-------+----------+
|SB10005|      56.0|
|SB10004|     500.0|
|SB10003|     330.0|
|SB10002|    8590.0|
|SB10001|   18900.0|
+-------+----------+
scala> // Create the DataFrame using API for the account summary records
scala> val acSummaryViaDFAPI =
acTransDF.groupBy("accNo").agg(sum("tranAmount") as "TransTotal")
acSummaryViaDFAPI: org.apache.spark.sql.DataFrame = [accNo: string,
TransTotal: double]
scala> // Show the first few records of the DataFrame
scala> acSummaryViaDFAPI.show
+-------+----------+
|  accNo|TransTotal|
+-------+----------+
|SB10005|      56.0|
|SB10004|     500.0|
|SB10003|     330.0|
|SB10002|    8590.0|
|SB10001|   18900.0|
+-------+----------+
```

In this code snippet, everything is very similar to the preceding section's code. The only difference is that, here, aggregations are used in the SQL queries as well as in the DataFrame API.

At the Python REPL prompt, try the following statements:

```
>>> from pyspark.sql import Row
>>> # Creation of the list from where the RDD is going to be created
>>> acTransList = ["SB10001,1000", "SB10002,1200",
"SB10001,8000","SB10002,400", "SB10003,300",
"SB10001,10000","SB10004,500","SB10005,56","SB10003,30","SB10002,7000",
```

```
    "SB10001,-100","SB10002,-10"]
>>> # Create the DataFrame
>>> acTransDF = sc.parallelize(acTransList).map(lambda trans:
trans.split(",")).map(lambda p: Row(accNo=p[0],
tranAmount=float(p[1]))).toDF()
>>> # Register temporary view in the DataFrame for using it in SQL
>>> acTransDF.createOrReplaceTempView("trans")
>>> # Use SQL to create another DataFrame containing the account summary
records
>>> acSummary = spark.sql("SELECT accNo, sum(tranAmount) as transTotal FROM
trans GROUP BY accNo")
>>> # Show the first few records of the DataFrame
>>> acSummary.show()
+-------+----------+
|  accNo|transTotal|
+-------+----------+
|SB10005|      56.0|
|SB10004|     500.0|
|SB10003|     330.0|
|SB10002|    8590.0|
|SB10001|   18900.0|
+-------+----------+
>>> # Create the DataFrame using API for the account summary records
>>> acSummaryViaDFAPI = acTransDF.groupBy("accNo").agg({"tranAmount":
"sum"}).selectExpr("accNo", "`sum(tranAmount)` as transTotal")
>>> # Show the first few records of the DataFrame
>>> acSummaryViaDFAPI.show()
+-------+----------+
|  accNo|transTotal|
+-------+----------+
|SB10005|      56.0|
|SB10004|     500.0|
|SB10003|     330.0|
|SB10002|    8590.0|
|SB10001|   18900.0|
+-------+----------+
```

In the DataFrame API for Python, there are some minor syntax differences as compared to its Scala counterpart.

Understanding multi-datasource joining with SparkSQL

In the previous chapter, the joining of multiple RDDs based on the key has been discussed. In this section, the same use case is implemented using Spark SQL. The use cases selected for elucidating the joining of multiple datasets using a key are given here.

The first dataset contains a retail banking master records summary with account number, first name, and last name. The second dataset contains the retail banking account balance with account number and balance amount. The key on both of the datasets is account number. Join the two datasets and create one dataset containing account number, first name, last name, and balance amount. From this report, pick up the top three accounts in terms of the balance amount.

In this section, the concept of joining data from multiple data sources is also demonstrated. First the DataFrames are created from two arrays. They are persisted in Parquet and JSON formats. Then they are read from the disk to form the DataFrames, and they are joined together.

At the Scala REPL prompt, try the following statements:

```scala
scala> // Define the case classes for using in conjunction with DataFrames
scala> case class AcMaster(accNo: String, firstName: String, lastName:
String)
defined class AcMaster
scala> case class AcBal(accNo: String, balanceAmount: Double)
defined class AcBal
scala> // Functions to convert the sequence of strings to objects defined
by the case classes
scala> def toAcMaster =  (master: Seq[String]) => AcMaster(master(0),
master(1), master(2))
toAcMaster: Seq[String] => AcMaster
scala> def toAcBal =  (bal: Seq[String]) => AcBal(bal(0),
bal(1).trim.toDouble)
toAcBal: Seq[String] => AcBal
scala> // Creation of the list from where the RDD is going to be created
scala> val acMasterList =
Array("SB10001,Roger,Federer","SB10002,Pete,Sampras",
"SB10003,Rafael,Nadal","SB10004,Boris,Becker", "SB10005,Ivan,Lendl")
acMasterList: Array[String] = Array(SB10001,Roger,Federer,
SB10002,Pete,Sampras, SB10003,Rafael,Nadal, SB10004,Boris,Becker,
SB10005,Ivan,Lendl)
scala> // Creation of the list from where the RDD is going to be created
scala> val acBalList = Array("SB10001,50000",
"SB10002,12000","SB10003,3000", "SB10004,8500", "SB10005,5000")
```

```
acBalList: Array[String] = Array(SB10001,50000, SB10002,12000,
SB10003,3000, SB10004,8500, SB10005,5000)
scala> // Create the DataFrame
scala> val acMasterDF =
sc.parallelize(acMasterList).map(_.split(",")).map(toAcMaster(_)).toDF()
acMasterDF: org.apache.spark.sql.DataFrame = [accNo: string, firstName:
string ... 1 more field]
scala> // Create the DataFrame
scala> val acBalDF =
sc.parallelize(acBalList).map(_.split(",")).map(toAcBal(_)).toDF()
acBalDF: org.apache.spark.sql.DataFrame = [accNo: string, balanceAmount:
double]
scala> // Persist the data of the DataFrame into a Parquet file
scala> acMasterDF.write.parquet("scala.master.parquet")
scala> // Persist the data of the DataFrame into a JSON file
scala> acBalDF.write.json("scalaMaster.json")
scala> // Read the data into a DataFrame from the Parquet file
scala> val acMasterDFFromFile = spark.read.parquet("scala.master.parquet")
acMasterDFFromFile: org.apache.spark.sql.DataFrame = [accNo: string,
firstName: string ... 1 more field]
scala> // Register temporary view in the DataFrame for using it in SQL
scala> acMasterDFFromFile.createOrReplaceTempView("master")
scala> // Read the data into a DataFrame from the JSON file
scala> val acBalDFFromFile = spark.read.json("scalaMaster.json")
acBalDFFromFile: org.apache.spark.sql.DataFrame = [accNo: string,
balanceAmount: double]
scala> // Register temporary view in the DataFrame for using it in SQL
scala> acBalDFFromFile.createOrReplaceTempView("balance")
scala> // Show the first few records of the DataFrame
scala> acMasterDFFromFile.show
+-------+---------+--------+
|  accNo|firstName|lastName|
+-------+---------+--------+
|SB10001|    Roger| Federer|
|SB10002|     Pete| Sampras|
|SB10003|   Rafael|   Nadal|
|SB10004|    Boris|  Becker|
|SB10005|     Ivan|   Lendl|
+-------+---------+--------+
scala> acBalDFFromFile.show
+-------+-------------+
|  accNo|balanceAmount|
+-------+-------------+
|SB10001|      50000.0|
|SB10002|      12000.0|
|SB10003|       3000.0|
|SB10004|       8500.0|
|SB10005|       5000.0|
```

```
+-------+-------------+
scala> // Use SQL to create another DataFrame containing the account detail
records
scala> val acDetail = spark.sql("SELECT master.accNo, firstName, lastName,
balanceAmount FROM master, balance WHERE master.accNo = balance.accNo ORDER
BY balanceAmount DESC")
acDetail: org.apache.spark.sql.DataFrame = [accNo: string, firstName:
string ... 2 more fields]
scala> // Show the first few records of the DataFrame
scala> acDetail.show
+-------+---------+--------+-------------+
|  accNo|firstName|lastName|balanceAmount|
+-------+---------+--------+-------------+
|SB10001|    Roger| Federer|      50000.0|
|SB10002|     Pete| Sampras|      12000.0|
|SB10004|    Boris|  Becker|       8500.0|
|SB10005|     Ivan|   Lendl|       5000.0|
|SB10003|   Rafael|   Nadal|       3000.0|
+-------+---------+--------+-------------+
```

Continuing from the same Scala REPL session, the following lines of code get the same result through the DataFrame API:

```
scala> // Create the DataFrame using API for the account detail records
scala> val acDetailFromAPI = acMasterDFFromFile.join(acBalDFFromFile,
acMasterDFFromFile("accNo") === acBalDFFromFile("accNo"),
"inner").sort($"balanceAmount".desc).select(acMasterDFFromFile("accNo"),
acMasterDFFromFile("firstName"), acMasterDFFromFile("lastName"),
acBalDFFromFile("balanceAmount"))
acDetailFromAPI: org.apache.spark.sql.DataFrame = [accNo: string,
firstName: string ... 2 more fields]
scala> // Show the first few records of the DataFrame
scala> acDetailFromAPI.show
+-------+---------+--------+-------------+
|  accNo|firstName|lastName|balanceAmount|
+-------+---------+--------+-------------+
|SB10001|    Roger| Federer|      50000.0|
|SB10002|     Pete| Sampras|      12000.0|
|SB10004|    Boris|  Becker|       8500.0|
|SB10005|     Ivan|   Lendl|       5000.0|
|SB10003|   Rafael|   Nadal|       3000.0|
+-------+---------+--------+-------------+
scala> // Use SQL to create another DataFrame containing the top 3 account
detail records
scala> val acDetailTop3 = spark.sql("SELECT master.accNo, firstName,
lastName, balanceAmount FROM master, balance WHERE master.accNo =
balance.accNo ORDER BY balanceAmount DESC").limit(3)
acDetailTop3: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =
```

```
[accNo: string, firstName: string ... 2 more fields]
scala> // Show the first few records of the DataFrame
scala> acDetailTop3.show
+-------+---------+--------+-------------+
|  accNo|firstName|lastName|balanceAmount|
+-------+---------+--------+-------------+
|SB10001|    Roger| Federer|      50000.0|
|SB10002|     Pete| Sampras|      12000.0|
|SB10004|    Boris|  Becker|       8500.0|
+-------+---------+--------+-------------+
```

The join type selected in the preceding section of the code is inner join. Instead of that, any other type of join can be used, either through the SQL query way or through the DataFrame API way. In this particular use case, it can be seen that the DataFrame API is becoming a bit clunky, while the SQL query looks very straightforward. The point here is that depending on the situation, in the application code, the SQL query way and the DataFrame API way can be mixed to produce the desired result. The DataFrame acDetailTop3 given in the following scripts is an example of that.

At the Python REPL prompt, try the following statements:

```
>>> from pyspark.sql import Row
>>> # Creation of the list from where the RDD is going to be created
>>> AcMaster = Row('accNo', 'firstName', 'lastName')
>>> AcBal = Row('accNo', 'balanceAmount')
>>> acMasterList = ["SB10001,Roger,Federer","SB10002,Pete,Sampras",
"SB10003,Rafael,Nadal","SB10004,Boris,Becker", "SB10005,Ivan,Lendl"]
>>> acBalList = ["SB10001,50000", "SB10002,12000","SB10003,3000",
"SB10004,8500", "SB10005,5000"]
>>> # Create the DataFrame
>>> acMasterDF = sc.parallelize(acMasterList).map(lambda trans:
trans.split(",")).map(lambda r: AcMaster(*r)).toDF()
>>> acBalDF = sc.parallelize(acBalList).map(lambda trans:
trans.split(",")).map(lambda r: AcBal(r[0], float(r[1]))).toDF()
>>> # Persist the data of the DataFrame into a Parquet file
>>> acMasterDF.write.parquet("python.master.parquet")
>>> # Persist the data of the DataFrame into a JSON file
>>> acBalDF.write.json("pythonMaster.json")
>>> # Read the data into a DataFrame from the Parquet file
>>> acMasterDFFromFile = spark.read.parquet("python.master.parquet")
>>> # Register temporary table in the DataFrame for using it in SQL
>>> acMasterDFFromFile.createOrReplaceTempView("master")
>>> # Register temporary table in the DataFrame for using it in SQL
>>> acBalDFFromFile = spark.read.json("pythonMaster.json")
>>> # Register temporary table in the DataFrame for using it in SQL
>>> acBalDFFromFile.createOrReplaceTempView("balance")
>>> # Show the first few records of the DataFrame
```

```
>>> acMasterDFFromFile.show()
+-------+---------+--------+
|  accNo|firstName|lastName|
+-------+---------+--------+
|SB10001|    Roger| Federer|
|SB10002|     Pete| Sampras|
|SB10003|   Rafael|   Nadal|
|SB10004|    Boris|  Becker|
|SB10005|     Ivan|   Lendl|
+-------+---------+--------+
>>> # Show the first few records of the DataFrame
>>> acBalDFFromFile.show()
+-------+-------------+
|  accNo|balanceAmount|
+-------+-------------+
|SB10001|      50000.0|
|SB10002|      12000.0|
|SB10003|       3000.0|
|SB10004|       8500.0|
|SB10005|       5000.0|
+-------+-------------+
>>> # Use SQL to create another DataFrame containing the account detail
records
>>> acDetail = spark.sql("SELECT master.accNo, firstName, lastName,
balanceAmount FROM master, balance WHERE master.accNo = balance.accNo ORDER
BY balanceAmount DESC")
>>> # Show the first few records of the DataFrame
>>> acDetail.show()
+-------+---------+--------+-------------+
|  accNo|firstName|lastName|balanceAmount|
+-------+---------+--------+-------------+
|SB10001|    Roger| Federer|      50000.0|
|SB10002|     Pete| Sampras|      12000.0|
|SB10004|    Boris|  Becker|       8500.0|
|SB10005|     Ivan|   Lendl|       5000.0|
|SB10003|   Rafael|   Nadal|       3000.0|
+-------+---------+--------+-------------+
>>> # Create the DataFrame using API for the account detail records
>>> acDetailFromAPI = acMasterDFFromFile.join(acBalDFFromFile,
acMasterDFFromFile.accNo ==
acBalDFFromFile.accNo).sort(acBalDFFromFile.balanceAmount,
ascending=False).select(acMasterDFFromFile.accNo,
acMasterDFFromFile.firstName, acMasterDFFromFile.lastName,
acBalDFFromFile.balanceAmount)
>>> # Show the first few records of the DataFrame
>>> acDetailFromAPI.show()
+-------+---------+--------+-------------+
|  accNo|firstName|lastName|balanceAmount|
```

```
+-------+---------+--------+------------+
|SB10001|    Roger| Federer|     50000.0|
|SB10002|     Pete| Sampras|     12000.0|
|SB10004|    Boris|  Becker|      8500.0|
|SB10005|     Ivan|   Lendl|      5000.0|
|SB10003|   Rafael|   Nadal|      3000.0|
+-------+---------+--------+------------+
>>> # Use SQL to create another DataFrame containing the top 3 account
detail records
>>> acDetailTop3 = spark.sql("SELECT master.accNo, firstName, lastName,
balanceAmount FROM master, balance WHERE master.accNo = balance.accNo ORDER
BY balanceAmount DESC").limit(3)
>>> # Show the first few records of the DataFrame
>>> acDetailTop3.show()
+-------+---------+--------+------------+
|  accNo|firstName|lastName|balanceAmount|
+-------+---------+--------+------------+
|SB10001|    Roger| Federer|     50000.0|
|SB10002|     Pete| Sampras|     12000.0|
|SB10004|    Boris|  Becker|      8500.0|
+-------+---------+--------+------------+
```

In the preceding sections, application of the RDD operations on DataFrame has been demonstrated. This shows the capability of Spark SQL to interoperate with the RDDs and vice versa. In the same way, SQL queries and the DataFrame API can be mixed in to have flexibility to use the easiest method of computation when solving real-world use cases in the applications.

Introducing datasets

The Spark programming paradigm has many abstractions to choose from when it comes to developing data processing applications. The fundamentals of Spark programming start with RDDs that can easily deal with unstructured, semi-structured, and structured data. The Spark SQL library offers highly optimized performance when processing structured data. This makes the basic RDDs look inferior in terms of performance. To fill this gap, from Spark 1.6 onwards, a new abstraction, named Dataset, was introduced that complements the RDD-based Spark programming model. It works pretty much the same way as RDD when it comes to Spark transformations and Spark actions, and at the same time, it is highly optimized like the Spark SQL. Dataset API provides strong compile-time type safety when it comes to writing programs and, because of that, the Dataset API is available only in Scala and Java.

The transaction banking use case discussed in the chapter covering the Spark programming model is taken up again here to elucidate the dataset-based programming model, because this programming model has a very close resemblance to RDD-based programming. The use case mainly deals with a set of banking transaction records and various processing done on those records to extract various information from it. The use case descriptions are not repeated here and it is not difficult to understand by looking at the comments and the code.

The following code snippet demonstrates the methods used to create Dataset, along with its usage, conversion of RDD to DataFrame, and conversion of DataFrame to dataset. The RDD to DataFrame conversion has already been discussed, but is captured here again to to keep the concepts in context. This is mainly to prove that various programming models in Spark and the data abstractions are highly interoperable.

At the Scala REPL prompt, try the following statements:

```
scala> // Define the case classes for using in conjunction with DataFrames
and Dataset
scala> case class Trans(accNo: String, tranAmount: Double)
defined class Trans
scala> // Creation of the list from where the Dataset is going to be
created using a case class.
scala> val acTransList = Seq(Trans("SB10001", 1000), Trans("SB10002",1200),
Trans("SB10003", 8000), Trans("SB10004",400), Trans("SB10005",300),
Trans("SB10006",10000), Trans("SB10007",500), Trans("SB10008",56),
Trans("SB10009",30),Trans("SB10010",7000), Trans("CR10001",7000),
Trans("SB10002",-10))
acTransList: Seq[Trans] = List(Trans(SB10001,1000.0),
Trans(SB10002,1200.0), Trans(SB10003,8000.0), Trans(SB10004,400.0),
Trans(SB10005,300.0), Trans(SB10006,10000.0), Trans(SB10007,500.0),
Trans(SB10008,56.0), Trans(SB10009,30.0), Trans(SB10010,7000.0),
Trans(CR10001,7000.0), Trans(SB10002,-10.0))
scala> // Create the Dataset
scala> val acTransDS = acTransList.toDS()
acTransDS: org.apache.spark.sql.Dataset[Trans] = [accNo: string,
tranAmount: double]
scala> acTransDS.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
```

```
|SB10009|       30.0|
|SB10010|     7000.0|
|CR10001|     7000.0|
|SB10002|      -10.0|
+-------+----------+
scala> // Apply filter and create another Dataset of good transaction
records
scala> val goodTransRecords = acTransDS.filter(_.tranAmount >
0).filter(_.accNo.startsWith("SB"))
goodTransRecords: org.apache.spark.sql.Dataset[Trans] = [accNo: string,
tranAmount: double]
scala> goodTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|     1000.0|
|SB10002|     1200.0|
|SB10003|     8000.0|
|SB10004|      400.0|
|SB10005|      300.0|
|SB10006|    10000.0|
|SB10007|      500.0|
|SB10008|       56.0|
|SB10009|       30.0|
|SB10010|     7000.0|
+-------+----------+
scala> // Apply filter and create another Dataset of high value transaction
records
scala> val highValueTransRecords = goodTransRecords.filter(_.tranAmount >
1000)
highValueTransRecords: org.apache.spark.sql.Dataset[Trans] = [accNo:
string, tranAmount: double]
scala> highValueTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     1200.0|
|SB10003|     8000.0|
|SB10006|    10000.0|
|SB10010|     7000.0|
+-------+----------+
scala> // The function that identifies the bad amounts
scala> val badAmountLambda = (trans: Trans) => trans.tranAmount <= 0
badAmountLambda: Trans => Boolean = <function1>
scala> // The function that identifies bad accounts
scala> val badAcNoLambda = (trans: Trans) => trans.accNo.startsWith("SB")
== false
badAcNoLambda: Trans => Boolean = <function1>
```

```
scala> // Apply filter and create another Dataset of bad amount records
scala> val badAmountRecords = acTransDS.filter(badAmountLambda)
badAmountRecords: org.apache.spark.sql.Dataset[Trans] = [accNo: string,
tranAmount: double]
scala> badAmountRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     -10.0|
+-------+----------+
scala> // Apply filter and create another Dataset of bad account records
scala> val badAccountRecords = acTransDS.filter(badAcNoLambda)
badAccountRecords: org.apache.spark.sql.Dataset[Trans] = [accNo: string,
tranAmount: double]
scala> badAccountRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
+-------+----------+
scala> // Do the union of two Dataset and create another Dataset
scala> val badTransRecords  = badAmountRecords.union(badAccountRecords)
badTransRecords: org.apache.spark.sql.Dataset[Trans] = [accNo: string,
tranAmount: double]
scala> badTransRecords.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10002|     -10.0|
|CR10001|    7000.0|
+-------+----------+
scala> // Calculate the sum
scala> val sumAmount = goodTransRecords.map(trans =>
trans.tranAmount).reduce(_ + _)
sumAmount: Double = 28486.0
scala> // Calculate the maximum
scala> val maxAmount = goodTransRecords.map(trans =>
trans.tranAmount).reduce((a, b) => if (a > b) a else b)
maxAmount: Double = 10000.0
scala> // Calculate the minimum
scala> val minAmount = goodTransRecords.map(trans =>
trans.tranAmount).reduce((a, b) => if (a < b) a else b)
minAmount: Double = 30.0
scala> // Convert the Dataset to DataFrame
scala> val acTransDF = acTransDS.toDF()
acTransDF: org.apache.spark.sql.DataFrame = [accNo: string, tranAmount:
double]
scala> acTransDF.show()
```

```
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
scala> // Use Spark SQL to find out invalid transaction records
scala> acTransDF.createOrReplaceTempView("trans")
scala> val invalidTransactions = spark.sql("SELECT accNo, tranAmount FROM
trans WHERE (accNo NOT LIKE 'SB%') OR tranAmount <= 0")
invalidTransactions: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
scala> invalidTransactions.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
scala> // Interoperability of RDD, DataFrame and Dataset
scala> // Create RDD
scala> val acTransRDD = sc.parallelize(acTransList)
acTransRDD: org.apache.spark.rdd.RDD[Trans] = ParallelCollectionRDD[206] at
parallelize at <console>:28
scala> // Convert RDD to DataFrame
scala> val acTransRDDtoDF = acTransRDD.toDF()
acTransRDDtoDF: org.apache.spark.sql.DataFrame = [accNo: string,
tranAmount: double]
scala> // Convert the DataFrame to Dataset with the type checking
scala> val acTransDFtoDS = acTransRDDtoDF.as[Trans]
acTransDFtoDS: org.apache.spark.sql.Dataset[Trans] = [accNo: string,
tranAmount: double]
scala> acTransDFtoDS.show()
+-------+----------+
|  accNo|tranAmount|
+-------+----------+
|SB10001|    1000.0|
|SB10002|    1200.0|
```

```
|SB10003|    8000.0|
|SB10004|     400.0|
|SB10005|     300.0|
|SB10006|   10000.0|
|SB10007|     500.0|
|SB10008|      56.0|
|SB10009|      30.0|
|SB10010|    7000.0|
|CR10001|    7000.0|
|SB10002|     -10.0|
+-------+----------+
```

It is very clear that the dataset-based programming has good applicability in many of the data processing use cases; at the same time, it has got high interoperability with other data processing abstractions within Spark itself.

> In the preceding code snippet, the DataFrame was converted to Dataset with a type specification `acTransRDDToDF.as[Trans]`. This type of conversion is really required when reading data from external data sources such as JSON, Avro, or Parquet files. That is when strong type checking is needed. Typically, structured data is read into DataFrame, and that can be converted to DataSet with strong type safety check like this in one shot: `spark.read.json("/transaction.json").as[Trans]`

If the Scala code snippets throughout this chapter are examined, when some of the methods are called on a DataFrame, instead of returning a DataFrame object, an object of type `org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]` is returned. This is the important relationship between DataFrame and dataset. In other words, DataFrame is a dataset of type `org.apache.spark.sql.Row`. If required, this object of type `org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]` can be explicitly converted to DataFrame using the `toDF()` method.

Too many choices confuse everybody. Here in the Spark programming model, the same problem is seen. But it is not as confusing as in many other programming paradigms. Whenever there is a need to process any kind of data with very high flexibility in terms of the data processing requirements and having the lowest API level control such as library development, the RDD-based programming model is ideal. Whenever there is a need to process structured data with flexibility for accessing and processing data with optimized performance across all the supported programming languages, the DataFrame-based Spark SQL programming model is ideal.

Whenever there is a need to process unstructured data with optimized performance requirements as well as compile-time type safety but not very complex Spark transformations and Spark actions usage requirements, the dataset-based programming model is ideal. At a data processing application development level, if the programming language of choice permits, it is better to use dataset and DataFrame to have better performance.

Understanding Data Catalogs

The previous sections of this chapter covered programming models with DataFrames and datasets. Both of these programming models can deal with structured data. The structured data comes with metadata or the data describing the structure of the data. Spark SQL provides a minimalist API known as Catalog API for data processing applications to query and use the metadata in the applications. The Catalog API exposes a catalog abstraction with many databases in it. For the regular SparkSession, it will have only one database, namely default. But if Spark is used with Hive, then the entire Hive meta store will be available through the Catalog API. The following code snippets demonstrate the usage of Catalog API in Scala and Python.

Continuing from the same Scala REPL prompt, try the following statements:

```scala
scala> // Get the catalog object from the SparkSession object
scala> val catalog = spark.catalog
catalog: org.apache.spark.sql.catalog.Catalog =
org.apache.spark.sql.internal.CatalogImpl@14b8a751
scala> // Get the list of databases
scala> val dbList = catalog.listDatabases()
dbList: org.apache.spark.sql.Dataset[org.apache.spark.sql.catalog.Database]
= [name: string, description: string ... 1 more field]
scala> // Display the details of the databases
scala> dbList.select("name", "description", "locationUri").show()
 +-------+----------------+--------------------+
 | name| description| locationUri|
 +-------+----------------+--------------------+
 |default|default database|file:/Users/RajT/...|
 +-------+----------------+--------------------+
 scala> // Display the details of the tables in the database
 scala> val tableList = catalog.listTables()
tableList: org.apache.spark.sql.Dataset[org.apache.spark.sql.catalog.Table]
= [name: string, database: string ... 3 more fields]
 scala> tableList.show()
 +-----+--------+-----------+---------+-----------+
 | name|database|description|tableType|isTemporary|
 +-----+--------+-----------+---------+-----------+
```

```
   |trans| null| null|TEMPORARY| true|
 +-----+--------+-----------+---------+-----------+
scala> // The above list contains the temporary view that was created in
the Dataset use case discussed in the previous section
// The views created in the applications can be removed from the database
using the Catalog API
scala> catalog.dropTempView("trans")
// List the available tables after dropping the temporary view
scala> val latestTableList = catalog.listTables()
latestTableList:
org.apache.spark.sql.Dataset[org.apache.spark.sql.catalog.Table] = [name:
string, database: string ... 3 more fields]
scala> latestTableList.show()
+----+--------+-----------+---------+-----------+
|name|database|description|tableType|isTemporary|
+----+--------+-----------+---------+-----------+
+----+--------+-----------+---------+-----------+
```

Similarly, the Catalog API can be used from Python code as well. Since the dataset example was not applicable in Python, the table list will be empty. At the Python REPL prompt, try the following statements:

```
>>> #Get the catalog object from the SparkSession object
>>> catalog = spark.catalog
>>> #Get the list of databases and their details.
>>> catalog.listDatabases()
  [Database(name='default', description='default database', locationUri='
   file:/Users/RajT/source-code/spark-source/spark-2.0/spark-warehouse')]
// Display the details of the tables in the database
>>> catalog.listTables()
>>> []
```

The Catalog API is very handy when writing data processing applications with the ability to process data dynamically, based on the contents in the meta store, especially when using it in conjunction with Hive.

References

For more information you can refer to:

- https://amplab.cs.berkeley.edu/wp-content/uploads/2015/03/SparkSQLSigmod2015.pdf
- http://pandas.pydata.org/

Summary

Spark SQL is a very highly useful library on top of the Spark core infrastructure. This library makes the Spark programming more inclusive to a wider group of programmers who are well versed with the imperative style of programming but not as competent in functional programming. Apart from this, Spark SQL is the best library to process structured data in the Spark family of data processing libraries. Spark SQL-based data processing application programs can be written with SQL-like queries or DSL style imperative programs of DataFrame API. This chapter has also demonstrated various strategies of mixing RDD and DataFrames, mixing SQL-like queries and DataFrame API. This gives amazing flexibility for the application developers to write data processing programs in the way they are most comfortable with, or that is more appropriate to the use cases, and at the same time, without compromising performance.

The Dataset API is as the next generation of programming model based on dataset in Spark, providing optimized performance and compile-time type safety.

The Catalog API comes as a very handy tool to process data dynamically, based on the contents of the meta store.

R is the language of data scientists. Till the support of R as a programming language in Spark SQL was available, major distributed data processing was not easy for them. Now, using R as a language of choice, they can seamlessly write distributed data processing applications as if they are using an R data frame from their individual machines. The next chapter is going to discuss the use of R to do data processing in Spark SQL.

4
Spark Programming with R

R is a popular statistical computing programming language used by many and freely available under the **General Public License** (**GNU**). R originated from the programming language S, created by John Chambers. R was developed by Ross Ihaka and Robert Gentleman. Many data scientists use R for their computing needs. R has inherent support for many statistical functions and many scalar data types, and has composite data structures for vectors, matrices, data frames, and more, for statistical computation. R is highly extensible and for that, external packages can be created. Once an external package is created, it has to be installed and loaded for any program to use it. A collection of such packages under a directory forms an R library. In other words, R comes with a set of base packages and additional packages that can be installed on top of it to form the required library for the desired computing needs. In addition to functions, datasets can also be packaged in R packages.

We will cover the following topics in this chapter:

- The need for SparkR
- Essentials of R
- Dataframes
- Aggregations
- Multi-datasource joins with SparkR

The need for SparkR

A plain base R installation cannot interact with Spark. The **SparkR** package exposes all the required objects and functions for R to talk to the Spark ecosystem. Compared to Scala, Java, and Python, the Spark programming in R is different and the SparkR package mainly exposes R API for DataFrame-based Spark SQL programming. At the moment, R cannot be used to manipulate the RDDs of Spark directly. So for all practical purposes, the R API for Spark has access to only Spark SQL abstractions. The Spark **MLlib** can also be programmed using R because Spark MLlib uses DataFrames.

How is SparkR going to help the data scientists to do better data processing? The base R installation mandates that all the data to be stored (or accessible) on the computer where R is installed. The data processing occurs on the single computer on which the R installation is available. Moreover, if the data size is more than the main memory available on the computer, R will not be able to do the required processing. With the SparkR package, there is access to a whole new world of a cluster of nodes for data storage and for doing data processing. With the help of SparkR package, R can be used to access the Spark DataFrames as well as R DataFrames.

It is very important to know the distinction between the two types of data frames, R Dataframes and Spark Dataframes. An R DataFrame is completely local and a data structure of the R language. A Spark DataFrame is a parallel collection of structured data managed by the Spark infrastructure.

An R DataFrame can be converted to a Spark DataFrame and a Spark DataFrame can be converted to an R DataFrame.

When a Spark DataFrame is converted to an R DataFrame, it should fit in the available memory of the computer. This conversion is a great feature and there is a need to do so. By converting an R DataFrame to a Spark DataFrame, the data can be distributed and processed in parallel. By converting a Spark DataFrame to an R DataFrame, a lot of computations, charting, and plotting that is done by other R functions can be done. In a nutshell, the SparkR package brings the power of distributed and parallel computing capabilities to R.

Often, when performing data processing with R, because of the sheer size of the data and the need to fit it into the main memory of the computer, the data processing is done in multiple batches and the results are consolidated to compute the final results. This kind of multi-batch processing can be completely avoided if Spark with R is used to process the data.

Often, reporting, charting, and plotting are done on the aggregated and summarized raw data. The raw data size can be huge and need not fit into one computer. In such cases, Spark with R can be used to process the entire raw data and finally, the aggregated and summarized data can be used to produce the reports, charts, or plots.

Because of the inability to process huge amounts of data and for doing data analysis with R, many times, ETL tools are made to use for doing the pre-processing or transformations on the raw data, and only in the final stage is the data analysis done using R. Because of Spark's ability to process data at scale, Spark with R can replace the entire ETL pipeline and do the desired data analysis with R.

Many R users use the **dplyr** R package for manipulating datasets in R. This package provides fast data manipulation capabilities with R DataFrames. Just like manipulating local R DataFrames, it can access data from some of the RDBMS tables too. Apart from these primitive data manipulation capabilities, it lacks many of the data processing features available in Spark. So Spark with R is a good alternative to packages such as dplyr.

The SparkR package is yet another R package, but that is not stopping anybody from using any of the R packages that are already being used. At the same time, it supplements the data processing capability of R manifold by making use of the huge data processing capabilities of Spark.

Basics of the R language

This is not in any way a guide to R programming. But, it is important to touch upon the basics of R as a language very briefly for the benefit of those who are not familiar with R to appreciate what is being covered in this chapter. A very basic introduction to the language features is covered here.

R comes with a few built-in data types to hold numerical values, character values, and boolean values. There are composite data structures available and the most important ones are, namely, vectors, lists, matrices, and data frames. A vector consists of ordered collection of values of a given type. A list is an ordered collection of elements that can be of different types. For example, a list can hold two vectors, of which one is a vector containing numerical values and the the other is a vector containing boolean values. A matrix is a two-dimensional data structure holding numerical values in rows and columns. A data frame is a two-dimensional data structure containing rows and columns, where columns can have different data types but a single column cannot hold different data types.

Code samples of using a variable (a special case of vector), a numeric vector, a character vector, a list, a matrix, a data frame, and assigning column names to a data frame are as follows. The variable names are given as self-descriptive as possible for the reader to understand without the help of additional explanation. The following code snippet run on a regular R REPL gives an idea of the data structures of R:

```
$ r
R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Warning: namespace 'SparkR' is not available and has been replaced
by .GlobalEnv when processing object 'goodTransRecords'
[Previously saved workspace restored]
>
> x <- 5
> x
[1] 5
> aNumericVector <- c(10,10.5,31.2,100)
> aNumericVector
[1]  10.0  10.5  31.2 100.0
> aCharVector <- c("apple", "orange", "mango")
> aCharVector
[1] "apple"  "orange" "mango"
> aBooleanVector <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
> aBooleanVector
[1]  TRUE FALSE  TRUE FALSE FALSE
> aList <- list(aNumericVector, aCharVector)
> aList
[[1]]
[1]  10.0  10.5  31.2 100.0
[[2]]
[1] "apple"  "orange" "mango"
```

```
> aMatrix <- matrix(c(100, 210, 76, 65, 34, 45),nrow=3,ncol=2,byrow = TRUE)
> aMatrix
     [,1] [,2]
[1,]  100  210
[2,]   76   65
[3,]   34   45
> bMatrix <- matrix(c(100, 210, 76, 65, 34, 45),nrow=3,ncol=2,byrow =
FALSE)
> bMatrix
     [,1] [,2]
[1,]  100   65
[2,]  210   34
[3,]   76   45
> ageVector <- c(21, 35, 52)
> nameVector <- c("Thomas", "Mathew", "John")
> marriedVector <- c(FALSE, TRUE, TRUE)
> aDataFrame <- data.frame(ageVector, nameVector, marriedVector)
> aDataFrame
  ageVector nameVector marriedVector
1        21     Thomas         FALSE
2        35     Mathew          TRUE
3        52       John          TRUE
> colnames(aDataFrame) <- c("Age","Name", "Married")
> aDataFrame
  Age    Name Married
1  21 Thomas   FALSE
2  35 Mathew    TRUE
3  52   John    TRUE
```

The main topic of discussion here is going to be revolving around data frames. Some of the functions that are commonly used with data frames are demonstrated here. All these commands are to be executed on the regular R REPL as a continuation of the session that executed the preceding code snippet:

```
> # Returns the first part of the data frame and return two rows
> head(aDataFrame,2)
  Age    Name Married
1  21 Thomas   FALSE
2  35 Mathew    TRUE

> # Returns the last part of the data frame and return two rows
> tail(aDataFrame,2)
  Age    Name Married
2  35 Mathew    TRUE
3  52   John    TRUE
> # Number of rows in a data frame
> nrow(aDataFrame)
[1] 3
```

```
> # Number of columns in a data frame
> ncol(aDataFrame)
[1] 3
> # Returns the first column of the data frame. The return value is a data
frame
> aDataFrame[1]
  Age
1  21
2  35
3  52
> # Returns the second column of the data frame. The return value is a data
frame
> aDataFrame[2]
     Name
1 Thomas
2 Mathew
3   John
> # Returns the named columns of the data frame. The return value is a data
frame
> aDataFrame[c("Age", "Name")]
  Age    Name
1  21 Thomas
2  35 Mathew
3  52   John
> # Returns the contents of the second column of the data frame as a
vector.
> aDataFrame[[2]]
[1] Thomas Mathew John
Levels: John Mathew Thomas
> # Returns the slice of the data frame by a row
> aDataFrame[2,]
  Age   Name Married
2  35 Mathew    TRUE
> # Returns the slice of the data frame by multiple rows
> aDataFrame[c(1,2),]
  Age   Name Married
1  21 Thomas   FALSE
2  35 Mathew    TRUE
```

DataFrames in R and Spark

When working with Spark using R, it is very easy to get confused with the DataFrame data structure. As mentioned earlier, it is there in R and in Spark SQL. The following code snippet deals with converting an R DataFrame to a Spark DataFrame and vice versa. This is going to be a very common operation when programming Spark with R. The following

code snippet is to be executed in the R REPL of Spark. From now on, all the references to the R REPL are with respect to the R REPL of Spark:

```
$ cd $SPARK_HOME
$ ./bin/sparkR

R version 3.2.2 (2015-08-14) -- "Fire Safety"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Previously saved workspace restored]

Launching java with spark-submit command /Users/RajT/source-code/spark-
source/spark-2.0/bin/spark-submit   "sparkr-shell"
/var/folders/nf/trtmyt9534z03kq8p8zgbnxh0000gn/T//RtmpmuRsTC/backend_port2d
121acef4
Using Spark's default log4j profile: org/apache/spark/log4j-
defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
16/07/16 21:08:50 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable

 Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version  2.0.1-SNAPSHOT
      /_/


 Spark context is available as sc, SQL context is available as sqlContext
During startup - Warning messages:
1: 'SparkR::sparkR.init' is deprecated.
```

```
Use 'sparkR.session' instead.
See help("Deprecated")
2: 'SparkR::sparkRSQL.init' is deprecated.
Use 'sparkR.session' instead.
See help("Deprecated")
>
> # faithful is a data set and the data frame that comes with base R
> # Obviously it is an R DataFrame
> head(faithful)
  eruptions waiting
1     3.600      79
2     1.800      54
3     3.333      74
4     2.283      62
5     4.533      85
6     2.883      55
> tail(faithful)
    eruptions waiting
267     4.750      75
268     4.117      81
269     2.150      46
270     4.417      90
271     1.817      46
272     4.467      74
> # Convert R DataFrame to Spark DataFrame
> sparkFaithful <- createDataFrame(faithful)
> head(sparkFaithful)
  eruptions waiting
1     3.600      79
2     1.800      54
3     3.333      74
4     2.283      62
5     4.533      85
6     2.883      55
> showDF(sparkFaithful)
+--------+-------+
|eruptions|waiting|
+--------+-------+
|      3.6|   79.0|
|      1.8|   54.0|
|    3.333|   74.0|
|    2.283|   62.0|
|    4.533|   85.0|
|    2.883|   55.0|
|      4.7|   88.0|
|      3.6|   85.0|
|     1.95|   51.0|
|     4.35|   85.0|
```

```
|    1.833|   54.0|
|    3.917|   84.0|
|      4.2|   78.0|
|     1.75|   47.0|
|      4.7|   83.0|
|    2.167|   52.0|
|     1.75|   62.0|
|      4.8|   84.0|
|      1.6|   52.0|
|     4.25|   79.0|
+---------+-------+
only showing top 20 rows
> # Try calling a SparkR function showDF() on an R DataFrame. The following
error message will be shown
> showDF(faithful)
Error in (function (classes, fdef, mtable)   :
  unable to find an inherited method for function 'showDF' for signature
'"data.frame"'
> # Convert the Spark DataFrame to an R DataFrame
> rFaithful <- collect(sparkFaithful)
> head(rFaithful)
  eruptions waiting
1     3.600      79
2     1.800      54
3     3.333      74
4     2.283      62
5     4.533      85
6     2.883      55
```

There is no complete compatibility and interoperability between an R DataFrame and a Spark DataFrame in terms of the supported functions.

> As a good practice, it is better to name the R DataFrame and Spark DataFrame with agreed conventions in R programs in order to have a distinction between the two different types. Not all the functions that are supported on R DataFrames are not supported on Spark DataFrames and vice versa. Always refer to the right version of the R API for Spark.

Those who use a lot of charting and plotting have to be extra careful while dealing with R DataFrames in conjunction with Spark DataFrames. The charting and plotting of R works with only R DataFrames. If there is a need to produce charts or plots with the data processed by Spark and available in Spark DataFrame, it has to be converted to an R DataFrame to proceed with the charting and plotting. The following code snippet will give an idea this. We will use the faithful dataset again for elucidation purposes in the R REPL of Spark:

```
head(faithful)
  eruptions waiting
1     3.600      79
2     1.800      54
3     3.333      74
4     2.283      62
5     4.533      85
6     2.883      55
> # Convert the faithful R DataFrame to Spark DataFrame
> sparkFaithful <- createDataFrame(faithful)
> # The Spark DataFrame sparkFaithful NOT producing a histogram
> hist(sparkFaithful$eruptions,main="Distribution of
Eruptions",xlab="Eruptions")
Error in hist.default(sparkFaithful$eruptions, main = "Distribution of
Eruptions",   :
  'x' must be numeric
> # The R DataFrame faithful producing a histogram
> hist(faithful$eruptions,main="Distribution of
Eruptions",xlab="Eruptions")
```

The figure here is used jut to demonstrate that the Spark DataFrame cannot be used to do charting and R DataFrame has to be used for the same:
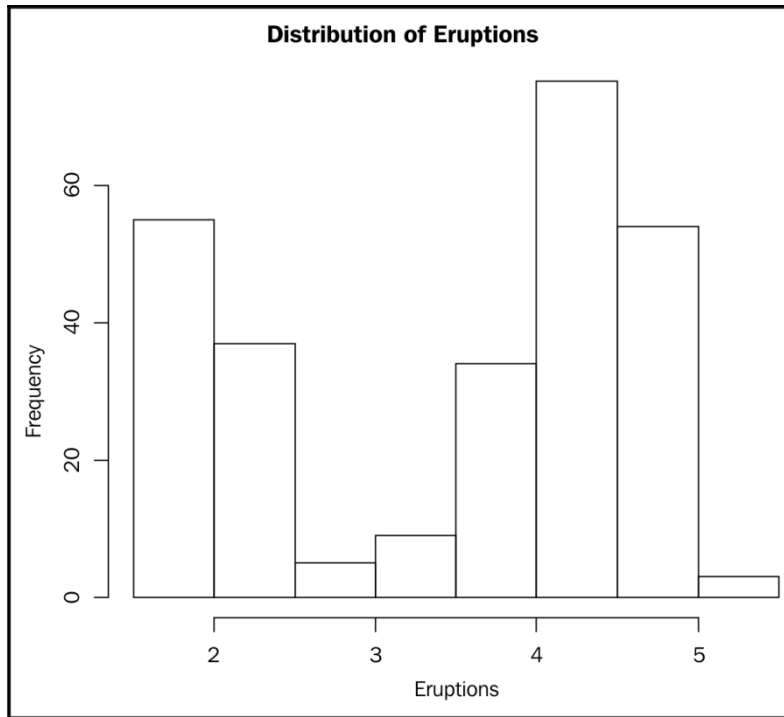
Figure 1

The charting and plotting library, when used with Spark DataFrame, gave an error because of the incompatibility of the data types.

> **TIP**
>
> The most important aspect to have in mind is that an R DataFrame is an in-memory resident data structure, while a Spark DataFrame is a parallel collection of datasets distributed across a cluster of nodes. So, all the functions that use R DataFrames need not work with Spark DataFrames and vice versa.

Let's revisit the bigger picture again, as given in *Figure 2*, to set the context and see what is being discussed here before getting into and taking up the use cases. In the previous chapter, the same subject was introduced by using the programming languages Scala and Python. In this chapter, the same set of use cases used in the Spark SQL programming will be implemented using R:
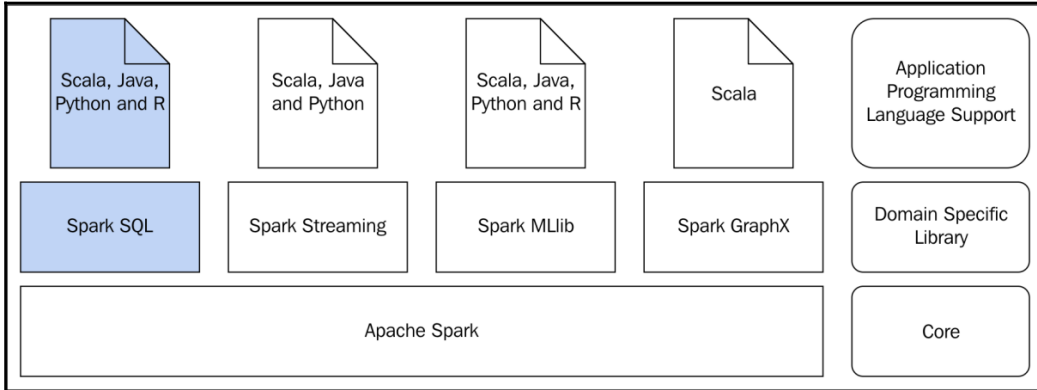
Figure 2

The use cases that are going to be discussed here will be demonstrating the ability to mix SQL queries with Spark programs in R. Multiple data sources will be chosen, data will be read from those sources using DataFrame, and uniform data access will be demonstrated.

Spark DataFrame programming with R

The use cases selected for elucidating the Spark SQL way of programming with DataFrame are given as follows:

- The transaction records are comma-separated values.
- Filter out only the good transaction records from the list. The account number should start with SB and the transaction amount should be greater than zero.
- Find all the high value transaction records with a transaction amount greater than 1000.
- Find all the transaction records where the account number is bad.
- Find all the transaction records where the transaction amount is less than or equal to zero.
- Find a combined list of all the bad transaction records.
- Find the total of all the transaction amounts.

- Find the maximum of all the transaction amounts.
- Find the minimum of all the transaction amounts.
- Find all the good account numbers.

This is exactly the same set of use cases that were used in the previous chapter, but here, the programming model is totally different. Here, the programming is done in R. Using this set of use cases, two types of programming model are demonstrated here. One is using the SQL queries and other is using DataFrame APIs.

> The data files needed for running the following code snippets are available from the same directory where the R code is kept.

In the following code snippets, data is read from files located in the filesystem. Since all these code snippets are executed from the R REPL of Spark, all the data files are to be kept in the $SPARK_HOME directory.

Programming with SQL

At the R REPL prompt, try the following statements:

```
> # TODO – Change the data directory location to the right location in the
system in which this program is being run
> DATA_DIR <- "/Users/RajT/Documents/CodeAndData/R/"
> # Read data from a JSON file to create DataFrame
>
> acTransDF <- read.json(paste(DATA_DIR, "TransList1.json", sep = ""))
> # Print the structure of the DataFrame
> print(acTransDF)
SparkDataFrame[AccNo:string, TranAmount:bigint]
> # Show sample records from the DataFrame
> showDF(acTransDF)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10001|      1000|
|SB10002|      1200|
|SB10003|      8000|
|SB10004|       400|
|SB10005|       300|
|SB10006|     10000|
|SB10007|       500|
|SB10008|        56|
```

```
|SB10009|         30|
|SB10010|       7000|
|CR10001|       7000|
|SB10002|        -10|
+-------+----------+
> # Register temporary view definition in the DataFrame for SQL queries
> createOrReplaceTempView(acTransDF, "trans")
> # DataFrame containing good transaction records using SQL
> goodTransRecords <- sql("SELECT AccNo, TranAmount FROM trans WHERE AccNo
like 'SB%' AND TranAmount > 0")
> # Register temporary table definition in the DataFrame for SQL queries

> createOrReplaceTempView(goodTransRecords, "goodtrans")
> # Show sample records from the DataFrame
> showDF(goodTransRecords)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10001|       1000|
|SB10002|       1200|
|SB10003|       8000|
|SB10004|        400|
|SB10005|        300|
|SB10006|      10000|
|SB10007|        500|
|SB10008|         56|
|SB10009|         30|
|SB10010|       7000|
+-------+----------+
> # DataFrame containing high value transaction records using SQL
> highValueTransRecords <- sql("SELECT AccNo, TranAmount FROM goodtrans
WHERE TranAmount > 1000")
> # Show sample records from the DataFrame
> showDF(highValueTransRecords)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10002|       1200|
|SB10003|       8000|
|SB10006|      10000|
|SB10010|       7000|
+-------+----------+
> # DataFrame containing bad account records using SQL
> badAccountRecords <- sql("SELECT AccNo, TranAmount FROM trans WHERE AccNo
NOT like 'SB%'")
> # Show sample records from the DataFrame
> showDF(badAccountRecords)
+-------+----------+
```

```
|  AccNo|TranAmount|
+-------+----------+
|CR10001|      7000|
+-------+----------+
> # DataFrame containing bad amount records using SQL
> badAmountRecords <- sql("SELECT AccNo, TranAmount FROM trans WHERE
TranAmount < 0")
> # Show sample records from the DataFrame
> showDF(badAmountRecords)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10002|       -10|
+-------+----------+
> # Create a DataFrame by taking the union of two DataFrames
> badTransRecords <- union(badAccountRecords, badAmountRecords)
> # Show sample records from the DataFrame
> showDF(badTransRecords)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|CR10001|      7000|
|SB10002|       -10|
+-------+----------+
> # DataFrame containing sum amount using SQL
> sumAmount <- sql("SELECT sum(TranAmount) as sum FROM goodtrans")
> # Show sample records from the DataFrame
> showDF(sumAmount)
+-----+
|  sum|
+-----+
|28486|
+-----+
> # DataFrame containing maximum amount using SQL
> maxAmount <- sql("SELECT max(TranAmount) as max FROM goodtrans")
> # Show sample records from the DataFrame
> showDF(maxAmount)
+-----+
|  max|
+-----+
|10000|
+-----+
> # DataFrame containing minimum amount using SQL
> minAmount <- sql("SELECT min(TranAmount)as min FROM goodtrans")
> # Show sample records from the DataFrame
> showDF(minAmount)
+---+
|min|
```

```
+---+
| 30|
+---+
> # DataFrame containing good account number records using SQL
> goodAccNos <- sql("SELECT DISTINCT AccNo FROM trans WHERE AccNo like
'SB%' ORDER BY AccNo")
> # Show sample records from the DataFrame
> showDF(goodAccNos)
+-------+
|  AccNo|
+-------+
|SB10001|
|SB10002|
|SB10003|
|SB10004|
|SB10005|
|SB10006|
|SB10007|
|SB10008|
|SB10009|
|SB10010|
+-------+
```

The retail banking transaction records come with account number, transaction amount are processed using SparkSQL to get the desired results of the use cases. Here is the summary of what the preceding script did:

- Unlike other programming languages supported with Spark, R doesn't have an RDD programming capability. So, instead of going with the construction of RDD from collections, the data is read from the JSON file containing the transaction records.
- A Spark DataFrame is created from the JSON file.
- A table is registered with the DataFrame with a name. This registered name of the table can be used in SQL statements.
- Then, all the other activities are issuing SQL statements using the SQL function from the SparkR package.
- The result of all these SQL statements is stored as Spark DataFrames, and showDF function is used to extract the values to the calling R program.
- The aggregate value calculations are also done through the SQL statements.

- The DataFrame contents are displayed in table format using the the `showDF` function of SparkR.
- A detailed view of the structure of the DataFrame is displayed using the print function. This is akin to the describe command of the database tables.

In the preceding R code, the style of programming is different as compared to the Scala code. That is because it is an R program. Using the SparkR library, the Spark features are being used. But the functions and other abstractions are not in a really different style.

> Throughout this chapter, there will be instances where DataFrames are used. It is very easy to get confused by which is the R DataFrame and which is the Spark DataFrame. Hence, care is taken to specifically mention by qualifying the DataFrame, such as R DataFrame and Spark DataFrame.

Programming with R DataFrame API

In this section, the code snippets will be run in the same R REPL. Like the preceding code snippets, initially, some DataFrame-specific basic commands are given. These are used regularly to see the contents and for doing some sanity tests on the DataFrame and its contents. These are commands that are typically used in the exploratory stage of the data analysis quite often to get more insight into the structure and contents of the underlying data.

At the R REPL prompt, try the following statements:

```
> # Read data from a JSON file to create DataFrame
> acTransDF <- read.json(paste(DATA_DIR, "TransList1.json", sep = ""))
> print(acTransDF)
SparkDataFrame[AccNo:string, TranAmount:bigint]
> # Show sample records from the DataFrame
> showDF(acTransDF)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10001|      1000|
|SB10002|      1200|
|SB10003|      8000|
|SB10004|       400|
|SB10005|       300|
|SB10006|     10000|
|SB10007|       500|
|SB10008|        56|
|SB10009|        30|
|SB10010|      7000|
```

```
|CR10001|      7000|
|SB10002|       -10|
+-------+----------+
> # DataFrame containing good transaction records using API
> goodTransRecordsFromAPI <- filter(acTransDF, "AccNo like 'SB%' AND
TranAmount > 0")
> # Show sample records from the DataFrame
> showDF(goodTransRecordsFromAPI)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10001|      1000|
|SB10002|      1200|
|SB10003|      8000|
|SB10004|       400|
|SB10005|       300|
|SB10006|     10000|
|SB10007|       500|
|SB10008|        56|
|SB10009|        30|
|SB10010|      7000|
+-------+----------+
> # DataFrame containing high value transaction records using API
> highValueTransRecordsFromAPI = filter(goodTransRecordsFromAPI,
"TranAmount > 1000")
> # Show sample records from the DataFrame
> showDF(highValueTransRecordsFromAPI)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10002|      1200|
|SB10003|      8000|
|SB10006|     10000|
|SB10010|      7000|
+-------+----------+
> # DataFrame containing bad account records using API
> badAccountRecordsFromAPI <- filter(acTransDF, "AccNo NOT like 'SB%'")
> # Show sample records from the DataFrame
> showDF(badAccountRecordsFromAPI)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|CR10001|      7000|
+-------+----------+
> # DataFrame containing bad amount records using API
> badAmountRecordsFromAPI <- filter(acTransDF, "TranAmount < 0")
> # Show sample records from the DataFrame
> showDF(badAmountRecordsFromAPI)
```

```
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10002|       -10|
+-------+----------+
> # Create a DataFrame by taking the union of two DataFrames
> badTransRecordsFromAPI <- union(badAccountRecordsFromAPI,
badAmountRecordsFromAPI)
> # Show sample records from the DataFrame
> showDF(badTransRecordsFromAPI)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|CR10001|      7000|
|SB10002|       -10|
+-------+----------+
> # DataFrame containing sum amount using API
> sumAmountFromAPI <- agg(goodTransRecordsFromAPI, sumAmount =
sum(goodTransRecordsFromAPI$TranAmount))
> # Show sample records from the DataFrame
> showDF(sumAmountFromAPI)
+---------+
|sumAmount|
+---------+
|    28486|
+---------+
> # DataFrame containing maximum amount using API
> maxAmountFromAPI <- agg(goodTransRecordsFromAPI, maxAmount =
max(goodTransRecordsFromAPI$TranAmount))
> # Show sample records from the DataFrame
> showDF(maxAmountFromAPI)
+---------+
|maxAmount|
+---------+
|    10000|
+---------+
> # DataFrame containing minimum amount using API
> minAmountFromAPI <- agg(goodTransRecordsFromAPI, minAmount =
min(goodTransRecordsFromAPI$TranAmount))
> # Show sample records from the DataFrame
> showDF(minAmountFromAPI)
+---------+
|minAmount|
+---------+
|       30|
+---------+
> # DataFrame containing good account number records using API
> filteredTransRecordsFromAPI <- filter(goodTransRecordsFromAPI, "AccNo
```

```
      like 'SB%'")
> accNosFromAPI <- select(filteredTransRecordsFromAPI, "AccNo")
> distinctAccNoFromAPI <- distinct(accNosFromAPI)
> sortedAccNoFromAPI <- arrange(distinctAccNoFromAPI, "AccNo")
> # Show sample records from the DataFrame
> showDF(sortedAccNoFromAPI)
+-------+
|  AccNo|
+-------+
|SB10001|
|SB10002|
|SB10003|
|SB10004|
|SB10005|
|SB10006|
|SB10007|
|SB10008|
|SB10009|
|SB10010|
+-------+
> # Persist the DataFrame into a Parquet file
> write.parquet(acTransDF, "r.trans.parquet")
> # Read the data from the Parquet file
> acTransDFFromFile <- read.parquet("r.trans.parquet")
> # Show sample records from the DataFrame
> showDF(acTransDFFromFile)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10007|       500|
|SB10008|        56|
|SB10009|        30|
|SB10010|      7000|
|CR10001|      7000|
|SB10002|       -10|
|SB10001|      1000|
|SB10002|      1200|
|SB10003|      8000|
|SB10004|       400|
|SB10005|       300|
|SB10006|     10000|
+-------+----------+
```

Here is the summary of what the preceding script did from a DataFrame API perspective:

- The DataFrame containing the superset of data used in the preceding section is used here.
- Filtering of the records is demonstrated next. Here, the most important aspect to notice is that the filter predicate is to be given exactly like the predicates in the SQL statements. Filters can not be chained.
- The aggregation methods are calculated next.
- The final statements in this set are doing the selection, filtering, choosing distinct records, and ordering.
- Finally, the transaction records are persisted in Parquet format, read from the Parquet store, and created a Spark DataFrame. More details on the persistence formats have been covered in the previous chapter and the concepts remain the same. Only the DataFrame API syntax is different.
- In this code snippet, the Parquet format data is stored in the current directory, from where the corresponding REPL is invoked. When it is run as a Spark program, the directory again will be the current directory from where the Spark submit is invoked.

The last few statements are about the persisting of the DataFrame contents into the media. If this is compared with the persistence mechanisms in the previous chapter with Scala and Python, here also it is done in similar ways.

Understanding aggregations in Spark R

In SQL, aggregation of data is very flexible. The same thing is true in Spark SQL too. Instead of running SQL statements on a single data source located in a single machine, here, Spark SQL can do the same on distributed data sources. In the chapter where RDD-based programming is covered, a MapReduce use case was discussed to do data aggregation and the same is being used here to demonstrate the aggregation capabilities of Spark SQL. In this section also, the use cases are approached in the SQL query way as well as in the DataFrame API way.

The use cases selected for elucidating the MapReduce kind of data processing are given here:

- The retail banking transaction records come with account number and transaction amount in comma-separated strings
- Find an account level summary of all the transactions to get the account balance

At the R REPL prompt, try the following statements:

```
> # Read data from a JSON file to create DataFrame
> acTransDFForAgg <- read.json(paste(DATA_DIR, "TransList2.json", sep =
""))
> # Register temporary view definition in the DataFrame for SQL queries
> createOrReplaceTempView(acTransDFForAgg, "transnew")
> # Show sample records from the DataFrame
> showDF(acTransDFForAgg)
+-------+----------+
|  AccNo|TranAmount|
+-------+----------+
|SB10001|      1000|
|SB10002|      1200|
|SB10001|      8000|
|SB10002|       400|
|SB10003|       300|
|SB10001|     10000|
|SB10004|       500|
|SB10005|        56|
|SB10003|        30|
|SB10002|      7000|
|SB10001|      -100|
|SB10002|       -10|
+-------+----------+
> # DataFrame containing account summary records using SQL
> acSummary <- sql("SELECT AccNo, sum(TranAmount) as TransTotal FROM
transnew GROUP BY AccNo")
> # Show sample records from the DataFrame
> showDF(acSummary)
+-------+----------+
|  AccNo|TransTotal|
+-------+----------+
|SB10001|     18900|
|SB10002|      8590|
|SB10003|       330|
|SB10004|       500|
|SB10005|        56|
+-------+----------+
> # DataFrame containing account summary records using API
> acSummaryFromAPI <- agg(groupBy(acTransDFForAgg, "AccNo"),
TranAmount="sum")
> # Show sample records from the DataFrame
> showDF(acSummaryFromAPI)
+-------+---------------+
|  AccNo|sum(TranAmount)|
+-------+---------------+
|SB10001|          18900|
```

```
|SB10002|          8590|
|SB10003|           330|
|SB10004|           500|
|SB10005|            56|
+-------+--------------+
```

In the R DataFrame API, there are some syntax differences as compared to its Scala or Python counterparts, mainly because this is a purely API-based programming model.

Understanding multi-datasource joins with SparkR

In the previous chapter, the joining of multiple DataFrames based on the key has been discussed. In this section, the same use case is implemented using R API of Spark SQL. The use cases selected for elucidating the join of multiple datasets using a key are given in the following section.

The first dataset contains a retail banking master records summary with the account number, first name, and last name. The second dataset contains the retail banking account balance with account number and balance amount. The key on both of the datasets is the account number. Join the two datasets and create one dataset containing the account number, first name, last name, and balance amount. From this report, pick up the top three accounts in terms of the balance amount.

The Spark DataFrames are created from persisted JSON files. Instead of the JSON files, it can be any supported data files. Then they are read from the disk to form the DataFrames and they are joined together.

At the R REPL prompt, try the following statements:

```
> # Read data from JSON file
> acMasterDF <- read.json(paste(DATA_DIR, "MasterList.json", sep = ""))
> # Show sample records from the DataFrame
> showDF(acMasterDF)
+-------+---------+--------+
|  AccNo|FirstName|LastName|
+-------+---------+--------+
|SB10001|    Roger| Federer|
|SB10002|     Pete| Sampras|
|SB10003|   Rafael|   Nadal|
|SB10004|    Boris|  Becker|
|SB10005|     Ivan|   Lendl|
+-------+---------+--------+
```

```
> # Register temporary view definition in the DataFrame for SQL queries
> createOrReplaceTempView(acMasterDF, "master")
> acBalDF <- read.json(paste(DATA_DIR, "BalList.json", sep = ""))
> # Show sample records from the DataFrame
> showDF(acBalDF)
+-------+---------+
|  AccNo|BalAmount|
+-------+---------+
|SB10001|    50000|
|SB10002|    12000|
|SB10003|     3000|
|SB10004|     8500|
|SB10005|     5000|
+-------+---------+

> # Register temporary view definition in the DataFrame for SQL queries
> createOrReplaceTempView(acBalDF, "balance")
> # DataFrame containing account detail records using SQL by joining
multiple DataFrame contents
> acDetail <- sql("SELECT master.AccNo, FirstName, LastName, BalAmount FROM
master, balance WHERE master.AccNo = balance.AccNo ORDER BY BalAmount
DESC")
> # Show sample records from the DataFrame
> showDF(acDetail)
+-------+---------+--------+---------+
|  AccNo|FirstName|LastName|BalAmount|
+-------+---------+--------+---------+
|SB10001|    Roger| Federer|    50000|
|SB10002|     Pete| Sampras|    12000|
|SB10004|    Boris|  Becker|     8500|
|SB10005|     Ivan|   Lendl|     5000|
|SB10003|   Rafael|   Nadal|     3000|
+-------+---------+--------+---------+

> # Persist data in the DataFrame into Parquet file
> write.parquet(acDetail, "r.acdetails.parquet")
> # Read data into a DataFrame by reading the contents from a Parquet file

> acDetailFromFile <- read.parquet("r.acdetails.parquet")
> # Show sample records from the DataFrame
> showDF(acDetailFromFile)
+-------+---------+--------+---------+
|  AccNo|FirstName|LastName|BalAmount|
+-------+---------+--------+---------+
|SB10002|     Pete| Sampras|    12000|
|SB10003|   Rafael|   Nadal|     3000|
|SB10005|     Ivan|   Lendl|     5000|
|SB10001|    Roger| Federer|    50000|
```

```
|SB10004|    Boris|  Becker|    8500|
+-------+---------+--------+---------+
```

Continuing from the same R REPL session, the following lines of code get the same result through the DataFrame API:

```
> # Change the column names
> acBalDFWithDiffColName <- selectExpr(acBalDF, "AccNo as AccNoBal",
"BalAmount")
> # Show sample records from the DataFrame
> showDF(acBalDFWithDiffColName)
+--------+---------+
|AccNoBal|BalAmount|
+--------+---------+
| SB10001|    50000|
| SB10002|    12000|
| SB10003|     3000|
| SB10004|     8500|
| SB10005|     5000|
+--------+---------+
> # DataFrame containing account detail records using API by joining
multiple DataFrame contents
> acDetailFromAPI <- join(acMasterDF, acBalDFWithDiffColName,
acMasterDF$AccNo == acBalDFWithDiffColName$AccNoBal)
> # Show sample records from the DataFrame
> showDF(acDetailFromAPI)
+-------+---------+--------+--------+---------+
|  AccNo|FirstName|LastName|AccNoBal|BalAmount|
+-------+---------+--------+--------+---------+
|SB10001|    Roger| Federer| SB10001|    50000|
|SB10002|     Pete| Sampras| SB10002|    12000|
|SB10003|   Rafael|   Nadal| SB10003|     3000|
|SB10004|    Boris|  Becker| SB10004|     8500|
|SB10005|     Ivan|   Lendl| SB10005|     5000|
+-------+---------+--------+--------+---------+
> # DataFrame containing account detail records using SQL by selecting
specific fields
> acDetailFromAPIRequiredFields <- select(acDetailFromAPI, "AccNo",
"FirstName", "LastName", "BalAmount")
> # Show sample records from the DataFrame
> showDF(acDetailFromAPIRequiredFields)
+-------+---------+--------+---------+
|  AccNo|FirstName|LastName|BalAmount|
+-------+---------+--------+---------+
|SB10001|    Roger| Federer|    50000|
|SB10002|     Pete| Sampras|    12000|
|SB10003|   Rafael|   Nadal|     3000|
```

```
|SB10004|    Boris|   Becker|    8500|
|SB10005|     Ivan|    Lendl|    5000|
+------+---------+--------+---------+
```

The join type selected in the preceding section of the code is inner join. Instead of that, any other type of join can be used, either through the SQL query way or through the DataFrame API way. One word of caution before using the join using DataFrame API is that the column names of both the Spark DataFrames have to be different to avoid ambiguity in the resultant Spark DataFrame. In this particular use case, it can be seen that the DataFrame API is becoming a bit difficult to deal with, while the SQL query way is looking very straightforward.

In the preceding sections, the R API for Spark SQL has been covered. In general, if possible, it is better to write the code using the SQL query way as much as possible. The DataFrame API is getting better, but it is not as flexible as in the other languages, such as Scala or Python.

Unlike the other chapters in this book, this is a self-contained one to introduce Spark to R programmers. All the use cases that are discussed in this chapter are run in the R REPL of Spark. But in real-world applications, this method is not ideal. The R commands have to be organized in script files and to be submitted to a Spark cluster to run. The easiest way is to use the already existing `$SPARK_HOME/bin/spark-submit <path to the R script file>` script, where the fully-qualified R filename is given with respect to the current directory from where the command is being invoked.

References

For more information refer to: `https://spark.apache.org/docs/latest/api/R/index.html`

Summary

A whirlwind tour of the R language was covered in this chapter, followed by a special mention about the need to have a distinction of understanding the difference between an R DataFrame and a Spark DataFrame. Then, basic Spark programming with R was covered using the same use cases of the previous chapters. R API for Spark was covered, and the use cases have been implemented using the SQL query way and DataFrame API way. This chapter helps data scientists understand the power of Spark and use it in their R applications, using the SparkR package that comes with Spark. This opens up the door of big data processing, using Spark with R to process structured data.

The subject of Spark-based data processing in various languages has been discussed, and it is time to focus on some data analysis with charting and plotting. Python comes with a lot of charting and plotting libraries that produce publication quality pictures. The next chapter will discuss charting and plotting with the data processed by Spark.

5

Spark Data Analysis with Python

The ultimate goal of processing data is to use the results for answering business questions. It is very important to understand the data that is being used to answer the business questions. To understand the data better, various tabulation methods, charting, and plotting techniques are used. Visual representation of the data reinforces the understanding of the underlying data. Because of this, data visualization is used extensively in data analysis.

There are different terms that are used in various publications to mean the analysis of data for answering business questions. Data analysis, data analytics, and business intelligence, are some of the ubiquitous terms floating around. This chapter is not going to delve into the discussion on the meaning, similarities, or differences of these terms. On the other hand, the focus is going to be on how to bridge the gap between two major activities typically done by data scientists or data analysts. The first one being data processing. The second one is the use of the processed data to do analysis with the help of charting and plotting. Data analysis is the forte of data analysts and data scientists. This chapter is going to focus on the usage of Spark and Python to process the data, and produce charts and plots.

In many data analysis use cases, a super-set of data is processed and the reduced resultant dataset is used for the data analysis. This is specifically valid in the case of big data analysis where a small set of processed data is used for analysis. Depending on the use case, for various data analysis needs, appropriate data processing is done as a prerequisite. Most of the use cases that are going to be covered in this chapter fall into this model, where the first step deals with the necessary data processing, and the second step deals with the charting and plotting required for the data analysis.

In typical data analysis use cases, the chain of activities involves an extensive and multi-staged **Extract**, **Transform**, and **Load** (**ETL**) pipeline ending with a data analysis platform or application. The end result of this chain of activities includes, but is not limited to, tables of summary data and various visual representations of the data in the form of charts and plots. Since Spark can process data from heterogeneous distributed data sources very effectively, the huge ETL pipeline that existed in legacy data analysis applications can be consolidated into self-contained applications that do the data processing and data analysis.

We will cover the following topics in this chapter:

- Charting and plotting libraries
- Setting up a dataset
- Capturing the high-level details of the data analysis use cases
- Various charts and plots

Charting and plotting libraries

Python is a programming language heavily used by data analysts and data scientists these days. There are numerous scientific and statistical data processing libraries available, as well as charting and plotting libraries, that can be used in Python programs. Python is also widely used as a programming language to develop data processing applications in Spark. This brings in a great flexibility to have a unified data processing and data analysis framework with Spark, Python and Python libraries, enabling us to do scientific and statistical processing, and charting and plotting. There are numerous such libraries that work with Python. Out of all those, the**NumPy** and **SciPy** libraries are being used here to do numerical, statistical, and scientific data processing. The **matplotlib** library is being used here to do the charting and plotting that produces 2D images.

> It is very important to make sure that the **NumPy**, **SciPy** and **matplotlib** Python libraries are working fine with the Python installation before attempting the code samples given in this chapter. This has to be tested and verified in isolation before using it in Spark applications.

The block diagram shown in *Figure 1* gives the overall structure of the application stack:
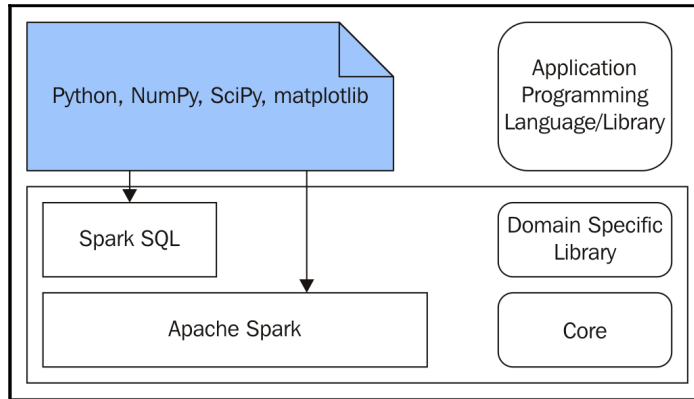
Figure 1

Setting up a dataset

There are many public Datasets available for the consumption of the general public that can be used for education, research, and development purposes. The MovieLens website lets users rate and personalize movie recommendations. GroupLens Research published the rating Datasets from MovieLens. These datasets are available for download from their website, `http://grouplens.org/datasets/movielens/`. In this chapter, the MovieLens 100K Dataset is being used to demonstrate the usage of distributed data processing with Spark in conjunction with Python, NumPy, SciPy, and matplotlib.

> On the GroupLens Research website for the dataset download, apart from the preceding dataset, there are more voluminous datasets such as MovieLens 1M dataset, MovieLens 10M dataset, MovieLens 20M dataset, and MovieLens latest datasets available for download. Once the reader is quite familiar with the programs and has achieved a sufficient level of comfort playing around with data, these additional datasets can be used by the reader to do their own analysis work to strengthen the knowledge acquired from this chapter.

The MovieLens 100K dataset has data in multiple files. The following are the ones that are going to be used in the data analysis use cases of this chapter:

- `u.user`: The demographic information about the users who have rated movies. The structure of the dataset is given as follows, reproduced as it is from the README file accompanying the dataset:
    - User ID
    - Age
    - Gender
    - Occupation
    - Zip code

- `u.item`: The information about the movies that are rated by the users. The structure of the dataset is given as follows, reproduced as it is from the README file accompanying the dataset:
    - Movie ID
    - Movie title
    - Release date
    - Video release date
    - IMDb URL
    - Unknown
    - Action
    - Adventure
    - Animation
    - Children's
    - Comedy
    - Crime
    - Documentary
    - Drama
    - Fantasy
    - Film-Noir
    - Horror
    - Musical
    - Mystery
    - Romance
    - Sci-Fi
    - Thriller

- War
- Western

Data analysis use cases

The following list captures the high-level details of the data analysis use cases. Most of the use cases are revolving around the creation of various charts and plots:

- Plot the age distribution of the users who have rated the movies using a histogram.
- Plot the age probability density chart of the users using the same data used to plot the histogram.
- Plot the summary of the age distribution data to find the minimum, 25th percentile, median, 75th percentile, and maximum ages of the users.
- Plot multiple charts or plots on the same figure to have a side-by-side comparison of the data.
- Create a bar chart capturing the top 10 occupations in terms of the number of users who have rated the movies.
- Create a stacked bar chart capturing the number of male and female users by their occupation who have rated the movies.
- Create a pie chart capturing the bottom 10 occupations in terms of the number of who have rated the movies.
- Create a donut chart capturing the top 10 zip codes in terms of the number of who have rated the movies.
- Using three occupation categories, create box plots capturing the summary statistics of the users who have rated the movies. All three box plots have to be drawn on a single figure to enable comparison.
- Create a bar chart capturing the number of movies by their genre.
- Create a scatter plot capturing the top 10 years in terms of the number of movies released in each year.
- Create a scatter plot capturing the top 10 years in terms of the number of movies released in each year. In this plot, instead of points in the plot, create circles with the area proportional to the number of movies released in that year.
- Create a line graph with two datasets with one dataset being the number of action movies released over the last 10 years and the other dataset being the number of drama movies released over the last 10 years to facilitate a comparison.

> **TIP**
>
> In all the preceding use cases, when it comes to implementation, Spark is used to process the data and prepare the required dataset. Once the required processed data is available in Spark DataFrame, it is collected into the driver program. In other words, the data is transferred from the distributed collection of Spark into a local collection, as tuples in the Python program, for charting and plotting. For charting and plotting, Python needs the data locally. It cannot use Spark DataFrames directly to do charting and plotting.

Charts and plots

This section is going to focus on creating various charts and plots to visually represent various aspects of the MovieLens 100K Dataset that are related to the use cases described in the preceding section. The charts and plots drawing process described throughout this chapter follows a pattern. Here are the important steps in that pattern of activities:

1. Read data from the data file using Spark.
2. Make the data available in a Spark DataFrame.
3. Apply the necessary data processing using DataFrame API.
4. The processing is mainly to make available only the minimal and required data for charting and plotting purposes.
5. Transfer the processed data from Spark DataFrame to the local Python collection object in the Spark Driver program.
6. Use the charting and plotting libraries to generate the figures using the data available in the Python collection objects.

Histogram

A histogram is generally used to show how a given numerical dataset is distributed over consecutive non-overlapping intervals of equal size. The interval or bin size is chosen based on the dataset. The bin or interval represents the ranges of data. In this use case, the dataset consists of the ages of the users. In this case it does not make sense to have a bin size of 100 as there will be only one bin and the entire dataset will fall into it. The height of the bars representing the bins indicates the frequency of data items in that bin or interval.

The following set of commands are used to bring up the Python REPL of Spark, followed by the programs to do the data processing, charting, and plotting:

```
$ cd $SPARK_HOME
$ ./bin/pyspark
>>> # Import all the required libraries
>>> from pyspark.sql import Row
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import pylab as P
>>> plt.rcdefaults()
>>> # TODO – The following location has to be changed to the
appropriate data file location
>>> dataDir =
"/Users/RajT/Documents/Writing/SparkForBeginners/SparkDataAnalysisWithPytho
n/Data/ml–100k/"
>>> # Create the DataFrame of the user dataset
>>> lines = sc.textFile(dataDir + "u.user")
>>> splitLines = lines.map(lambda l: l.split("|"))
>>> usersRDD = splitLines.map(lambda p: Row(id=p[0], age=int(p[1]),
gender=p[2], occupation=p[3], zipcode=p[4]))
>>> usersDF = spark.createDataFrame(usersRDD)
>>> usersDF.createOrReplaceTempView("users")
>>> usersDF.show()
    +---+------+---+-------------+-------+
    |age|gender| id|   occupation|zipcode|
    +---+------+---+-------------+-------+
    | 24|     M|  1|   technician|  85711|
    | 53|     F|  2|        other|  94043|
    | 23|     M|  3|       writer|  32067|
    | 24|     M|  4|   technician|  43537|
    | 33|     F|  5|        other|  15213|
    | 42|     M|  6|    executive|  98101|
    | 57|     M|  7|administrator|  91344|
    | 36|     M|  8|administrator|  05201|
    | 29|     M|  9|      student|  01002|
    | 53|     M| 10|       lawyer|  90703|
    | 39|     F| 11|        other|  30329|
    | 28|     F| 12|        other|  06405|
    | 47|     M| 13|     educator|  29206|
    | 45|     M| 14|    scientist|  55106|
    | 49|     F| 15|     educator|  97301|
    | 21|     M| 16|entertainment|  10309|
    | 30|     M| 17|   programmer|  06355|
    | 35|     F| 18|        other|  37212|
    | 40|     M| 19|    librarian|  02138|
    | 42|     F| 20|    homemaker|  95660|
```

```
      +---+------+---+------------+-------+
      only showing top 20 rows
>>> # Create the DataFrame of the user dataset with only one column age
>>> ageDF = spark.sql("SELECT age FROM users")
>>> ageList = ageDF.rdd.map(lambda p: p.age).collect()
>>> ageDF.describe().show()
      +-------+------------------+
      |summary|               age|
      +-------+------------------+
      |  count|               943|
      |   mean| 34.05196182396607|
      | stddev|12.186273150937206|
      |    min|                 7|
      |    max|                73|
      +-------+------------------+

>>> # Age distribution of the users
>>> plt.hist(ageList)
>>> plt.title("Age distribution of the users\n")
>>> plt.xlabel("Age")
>>> plt.ylabel("Number of users")
>>> plt.show(block=False)
```

In the preceding section, the user dataset was read line by line to form the RDD. From the RDD, a Spark DataFrame was created. Using Spark SQL, another Spark DataFrame was created containing only the age column. The summary of that Spark DataFrame was displayed to show the summary statistics of the contents; the contents were collected into a local Python collection object. Using the collected data, a histogram of the age column was plotted, as given in *Figure 2*:
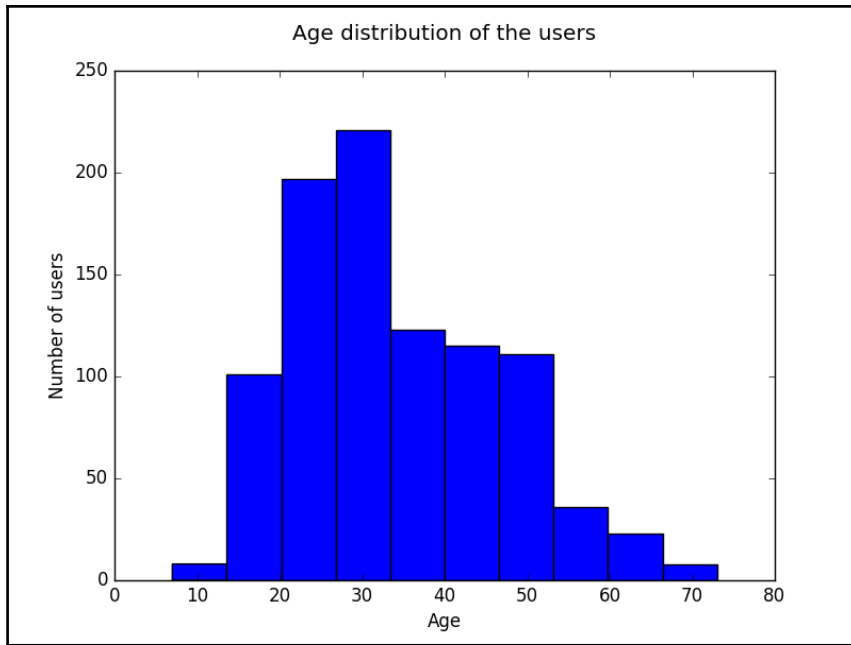
Figure 2

Density plot

There is another plot that is very close to a histogram. It is the density plot. Whenever there is a finite data sample with a need to estimate the probability density function of a random variable, density plots are used heavily. A histogram doesn't show when data smooths out or when there is continuity in the data points. For that purpose, density plots are used.

> Since a histogram and density plot are used for similar purposes, but show different behavior for the same data, generally, a histogram and density plot are used side by side in many applications.

Figure 3 is a density plot drawn for the same dataset that is used to plot the histogram.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> # Draw a density plot
>>> from scipy.stats import gaussian_kde
```

```
>>> density = gaussian_kde(ageList)
>>> xAxisValues = np.linspace(0,100,1000)
>>> density.covariance_factor = lambda : .5
>>> density._compute_covariance()
>>> plt.title("Age density plot of the users\n")
>>> plt.xlabel("Age")
>>> plt.ylabel("Density")
>>> plt.plot(xAxisValues, density(xAxisValues))
>>> plt.show(block=False)
```
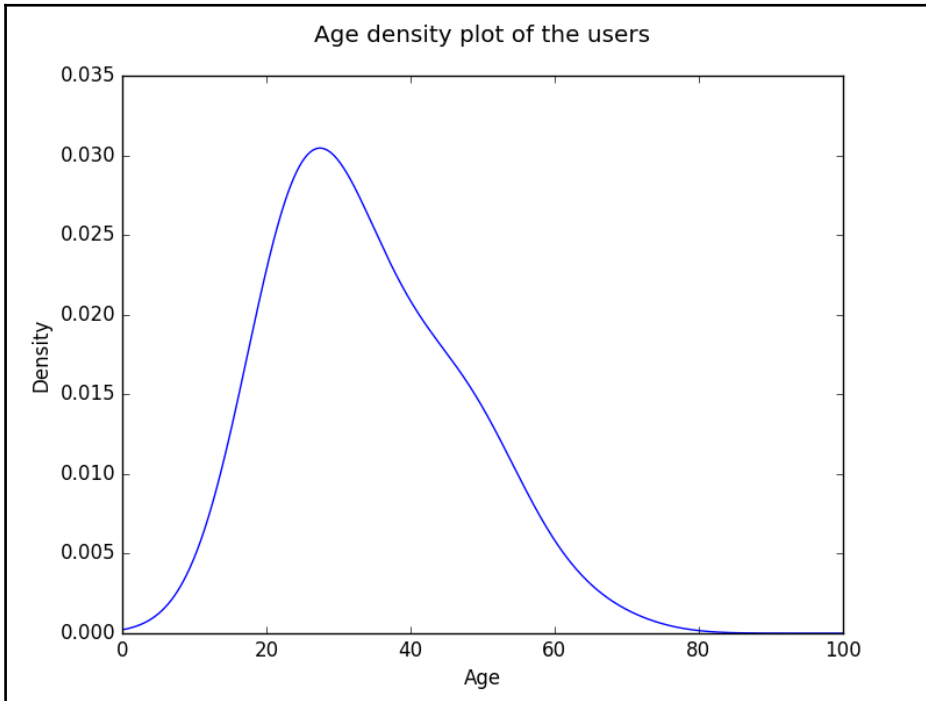

Figure 3

In the preceding section, the same Spark DataFrame that was created containing only the age column was used and the contents were collected into a local Python collection object. Using the collected data, a density plot of the age column was plotted as given in *Figure 3*, with the line space from 0 to 100 representing the age.

If multiple charts or plots are to be looked at side by side, the **matplotlib** library provides ways to do that. Figure 4 shows a histogram and a box plot side by side.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> # The following example demonstrates the creation of multiple
diagrams
    in one figure
>>> # There are two plots on one row
>>> # The first one is the histogram of the distribution
>>> # The second one is the boxplot containing the summary of the
    distribution
>>> plt.subplot(121)
>>> plt.hist(ageList)
>>> plt.title("Age distribution of the users\n")
>>> plt.xlabel("Age")
>>> plt.ylabel("Number of users")
>>> plt.subplot(122)
>>> plt.title("Summary of distribution\n")
>>> plt.xlabel("Age")
>>> plt.boxplot(ageList, vert=False)
>>> plt.show(block=False)
```
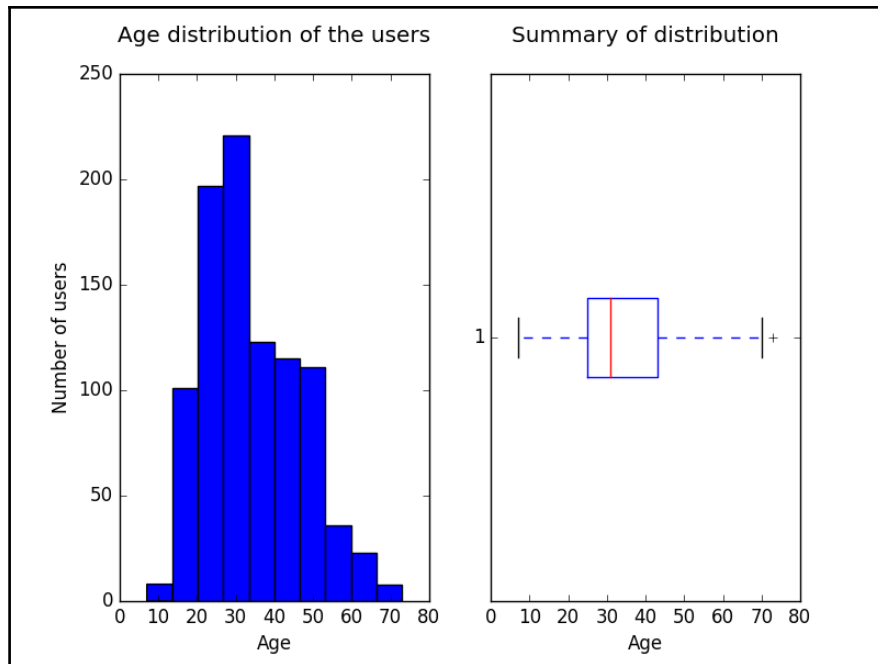

Figure 4

In the preceding section, the same Spark DataFrame that was created containing only the age column was used, and the contents were collected into a local Python collection object. Using the collected data, a histogram of the age column was plotted along with a box plot containing indicators for the minimum, 25th percentile, median, 75th percentile, and maximum values, as given in *Figure 4*. When drawing multiple charts or plots in one figure, for a way to control the layout, look at the method call `plt.subplot(121)`. This is talking about the selection of the plot laid out in one row and two columns, and selects the first one. In the same way, `plt.subplot(122)` talks about the selection of the plot laid out in one row and two columns, and selects the second one.

Bar chart

Bar charts can be drawn in different ways. The most common one is where the bars are standing vertically on the *X* axis. Another variation is where the bars are drawn on the *Y* axis and have the bars laid out horizontally. *Figure 5* shows a horizontal bar chart.

> It is very common to get confused between a histogram and bar chart. The important difference is that a histogram is used to plot continuous but finite numerical values, but a bar chart is used to represent categorical data.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> occupationsTop10 = spark.sql("SELECT occupation, count(occupation) as
usercount FROM users GROUP BY occupation ORDER BY usercount DESC LIMIT 10")
>>> occupationsTop10.show()
+-------------+---------+
|   occupation|usercount|
+-------------+---------+
|      student|      196|
|        other|      105|
|     educator|       95|
|administrator|       79|
|     engineer|       67|
|   programmer|       66|
|    librarian|       51|
|       writer|       45|
|    executive|       32|
|    scientist|       31|
+-------------+---------+
>>> occupationsTop10Tuple = occupationsTop10.rdd.map(lambda p:
(p.occupation,p.usercount)).collect()
>>> occupationsTop10List, countTop10List = zip(*occupationsTop10Tuple)
>>> occupationsTop10Tuple
```

```
>>> # Top 10 occupations in terms of the number of users having that
occupation who have rated movies
>>> y_pos = np.arange(len(occupationsTop10List))
>>> plt.barh(y_pos, countTop10List, align='center', alpha=0.4)
>>> plt.yticks(y_pos, occupationsTop10List)
>>> plt.xlabel('Number of users')
>>> plt.title('Top 10 user types\n')
>>> plt.gcf().subplots_adjust(left=0.15)
>>> plt.show(block=False)
```
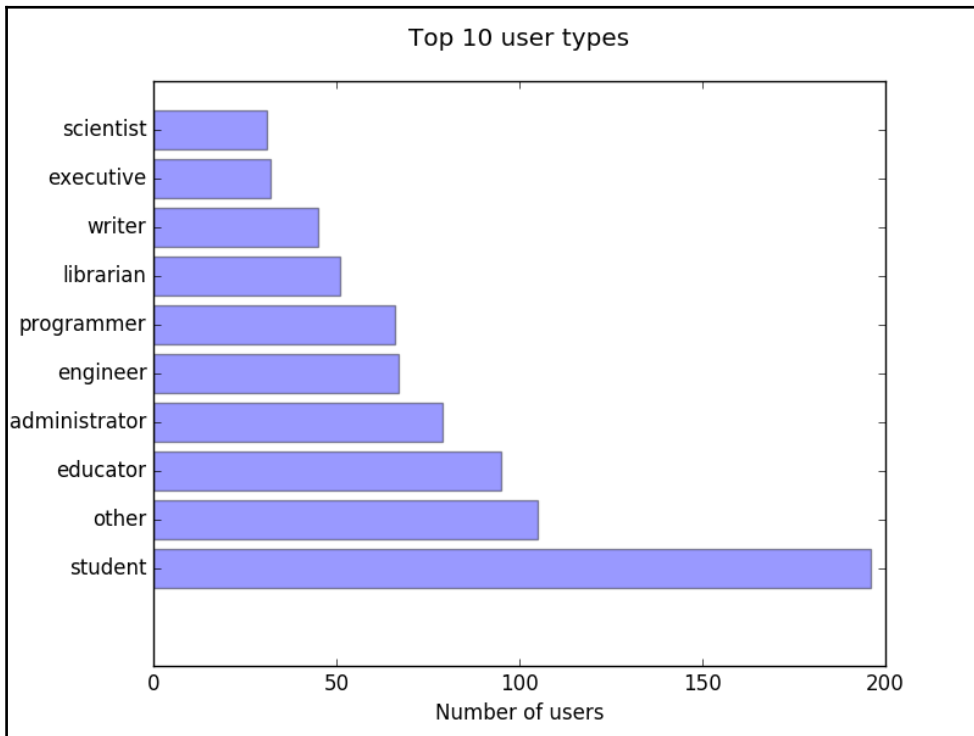

Figure 5

In the preceding section, a Spark DataFrame was created containing the top 10 occupations of the users in terms of the number of users who have rated movies. The data was collected into a Python collection object to plot the bar chart.

Stacked bar chart

The bar chart that was drawn in the preceding section gives the top 10 user occupations in terms of the number of users. But that does not give details about how that number is made up in terms of the gender of the users. In this kind of situation, it is good to use a stacked bar chart with each bar showing the counts by gender. *Figure 6* shows a stacked bar chart.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> occupationsGender = spark.sql("SELECT occupation, gender FROM
users")
>>> occupationsGender.show()
+-------------+------+
|   occupation|gender|
+-------------+------+
|   technician|     M|
|        other|     F|
|       writer|     M|
|   technician|     M|
|        other|     F|
|    executive|     M|
|administrator|     M|
|administrator|     M|
|      student|     M|
|       lawyer|     M|
|        other|     F|
|        other|     F|
|     educator|     M|
|    scientist|     M|
|     educator|     F|
|entertainment|     M|
|    programmer|    M|
|        other|     F|
|     librarian|    M|
|     homemaker|    F|
+-------------+------+
only showing top 20 rows
>>> occCrossTab = occupationsGender.stat.crosstab("occupation",
"gender")
>>> occCrossTab.show()
+----------------+---+---+
|occupation_gender|  M|  F|
+----------------+---+---+
|        scientist| 28|  3|
|          student|136| 60|
|           writer| 26| 19|
|         salesman|  9|  3|
|          retired| 13|  1|
```

```
|     administrator|  43|  36|
|        programmer|  60|   6|
|            doctor|   7|   0|
|         homemaker|   1|   6|
|         executive|  29|   3|
|          engineer|  65|   2|
|     entertainment|  16|   2|
|         marketing|  16|  10|
|        technician|  26|   1|
|            artist|  15|  13|
|          librarian|  22|  29|
|            lawyer|  10|   2|
|           educator|  69|  26|
|         healthcare|   5|  11|
|              none|   5|   4|
+------------------+---+---+
only showing top 20 rows
>>> occupationsCrossTuple = occCrossTab.rdd.map(lambda p:
(p.occupation_gender,p.M, p.F)).collect()
>>> occList, mList, fList = zip(*occupationsCrossTuple)
>>> N = len(occList)
>>> ind = np.arange(N)      # the x locations for the groups
>>> width = 0.75            # the width of the bars
>>> p1 = plt.bar(ind, mList, width, color='r')
>>> p2 = plt.bar(ind, fList, width, color='y', bottom=mList)
>>> plt.ylabel('Count')
>>> plt.title('Gender distribution by occupation\n')
>>> plt.xticks(ind + width/2., occList, rotation=90)
>>> plt.legend((p1[0], p2[0]), ('Male', 'Female'))
>>> plt.gcf().subplots_adjust(bottom=0.25)
>>> plt.show(block=False)
```
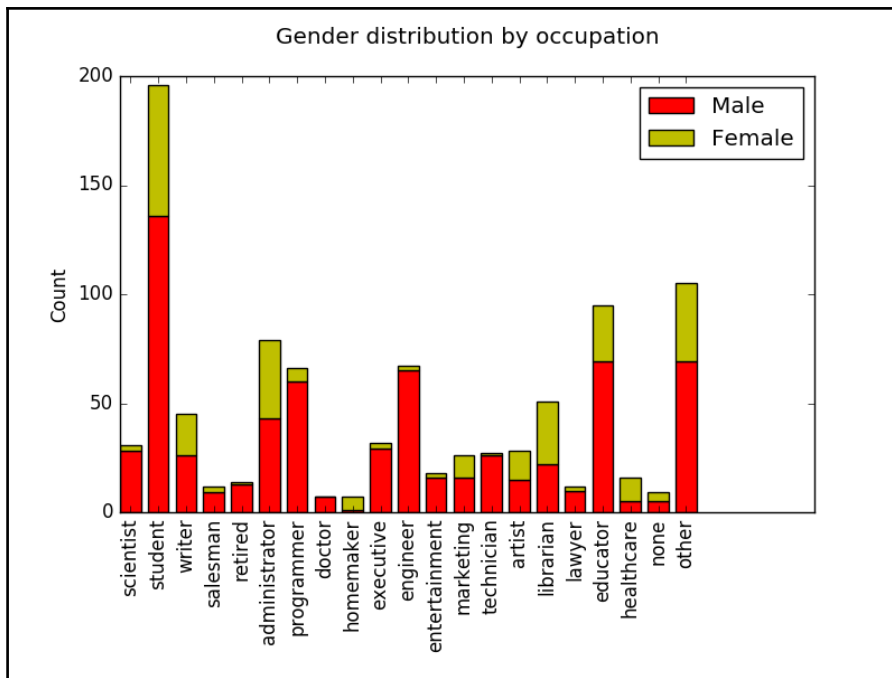
Figure 6

In the preceding section, a Spark DataFrame was created containing only the occupation and gender columns. A cross-tab operation was done on that to produce another Spark DataFrame, which produced columns for occupation, male user count, and female user count. In the first Spark DataFrame, containing the occupation and gender columns, both are non-numerical columns and, because of that, it doesn't make sense to draw as chart or plot based on that data. But if a cross-tab operation is done on these two column values, for every distinct occupation field, then the counts of values of the gender column will be available. In this way, the occupation field becomes a categorical variable and it makes sense to draw a bar chart with the data. Since there are only two gender values in this data, it makes sense to have a stacked bar chart to see the total as well as the proportions of male and female user counts in each occupation category.

There are many statistical and mathematical functions available within DataFrames in Spark. The cross-tab operation on a Spark DataFrame comes in very handy in this kind of situation. With huge datasets, the cross-tab operation can become very processor-intensive and time consuming, but the distributed processing capabilities of Spark are a great help in this kind of situation.

Spark SQL comes with lots of mathematical and statistical data processing capabilities. The preceding sections used the `describe().show()` method on the `SparkDataFrame` objects. In those Spark DataFrames, the preceding method acted on the available numeric columns. There will be situations where there are multiple numeric columns and, in those situations, the preceding method has the ability to pick and choose the desired columns for getting the summary statistics. Similarly, there are methods to find covariance, correlation, and so on, on the data from the Spark DataFrame. The following code snippet demonstrates these methods:

```
>>> occCrossTab.describe('M', 'F').show()
+-------+-----------------+-----------------+
|summary|                M|                F|
+-------+-----------------+-----------------+
|  count|               21|               21|
|   mean|31.904761904761905|             13.0|
| stddev|31.595516200735347|15.491933384829668|
|    min|                1|                0|
|    max|              136|               60|
+-------+-----------------+-----------------+
>>> occCrossTab.stat.cov('M', 'F')
  381.15
>>> occCrossTab.stat.corr('M', 'F')
  0.7416099517313641
```

Pie chart

If there is a need to visually represent a dataset to explain the whole-part relationship, a pie chart is very commonly used. *Figure 7* shows a pie chart.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> occupationsBottom10 = spark.sql("SELECT occupation,
count(occupation) as usercount FROM users GROUP BY occupation ORDER BY
usercount LIMIT 10")
>>> occupationsBottom10.show()
+-------------+---------+
|   occupation|usercount|
+-------------+---------+
```

```
|    homemaker|       7|
|       doctor|       7|
|         none|       9|
|     salesman|      12|
|       lawyer|      12|
|      retired|      14|
|   healthcare|      16|
|entertainment|      18|
|    marketing|      26|
|    technician|      27|
+-------------+--------+
>>> occupationsBottom10Tuple = occupationsBottom10.rdd.map(lambda p:
(p.occupation,p.usercount)).collect()
>>> occupationsBottom10List, countBottom10List =
zip(*occupationsBottom10Tuple)
>>> # Bottom 10 occupations in terms of the number of users having that
occupation who have rated movies
>>> explode = (0, 0, 0, 0,0.1,0,0,0,0,0.1)
>>> plt.pie(countBottom10List, explode=explode,
labels=occupationsBottom10List, autopct='%1.1f%%', shadow=True,
startangle=90)
>>> plt.title('Bottom 10 user types\n')
>>> plt.show(block=False)
```
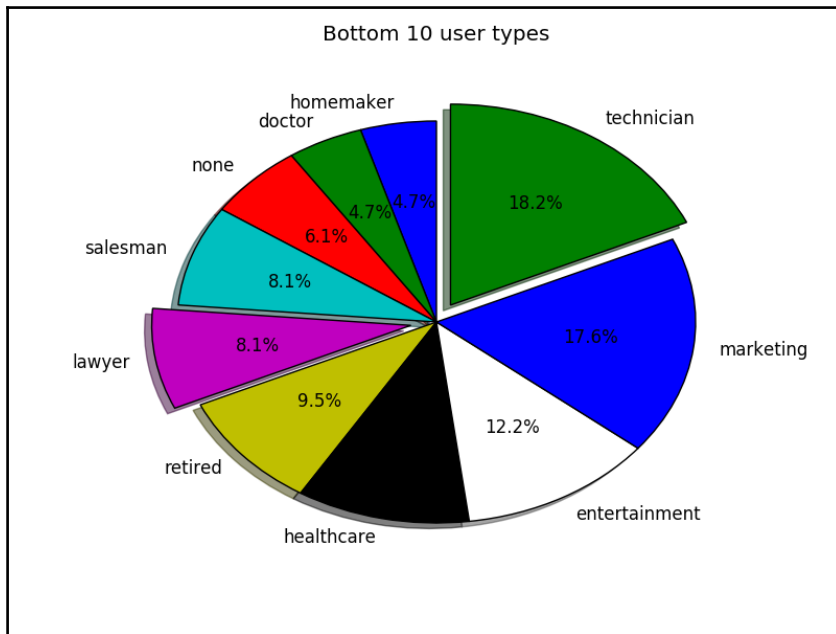

Figure 7

In the preceding section, a Spark DataFrame was created containing the bottom 10 occupations of the users in terms of the number of users who have rated movies. The data was collected into a Python collection object to plot the pie chart.

Donut chart

Pie charts can be drawn in different forms. One such form, the donut chart, is often used these days. Figure 8 shows this donut chart variation of the pie chart.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> zipTop10 = spark.sql("SELECT zipcode, count(zipcode) as usercount
FROM users GROUP BY zipcode ORDER BY usercount DESC LIMIT 10")
>>> zipTop10.show()
    +-------+---------+
    |zipcode|usercount|
    +-------+---------+
    |  55414|        9|
    |  55105|        6|
    |  20009|        5|
    |  55337|        5|
    |  10003|        5|
    |  55454|        4|
    |  55408|        4|
    |  27514|        4|
    |  11217|        3|
    |  14216|        3|
    +-------+---------+
>>> zipTop10Tuple = zipTop10.rdd.map(lambda p:
(p.zipcode,p.usercount)).collect()
>>> zipTop10List, countTop10List = zip(*zipTop10Tuple)
>>> # Top 10 zipcodes in terms of the number of users living in that
zipcode who have rated movies
>>> explode = (0.1, 0, 0, 0,0,0,0,0,0,0)  # explode a slice if required
>>> plt.pie(countTop10List, explode=explode, labels=zipTop10List,
autopct='%1.1f%%', shadow=True)
>>> #Draw a circle at the center of pie to make it look like a donut
>>> centre_circle = plt.Circle((0,0),0.75,color='black',
fc='white',linewidth=1.25)
>>> fig = plt.gcf()
>>> fig.gca().add_artist(centre_circle)
>>> # The aspect ratio is to be made equal. This is to make sure that
pie chart is coming perfectly as a circle.
>>> plt.axis('equal')
>>> plt.text(- 0.25,0,'Top 10 zip codes')
>>> plt.show(block=False)
```
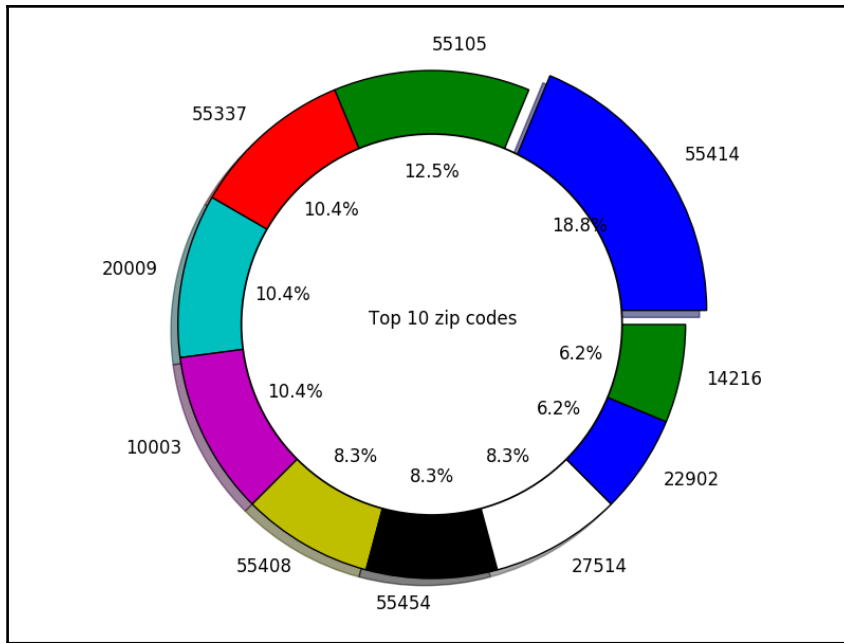
Figure 8

In the preceding section, a Spark DataFrame was created containing the top 10 zip codes of the users in terms of the number of users who live in that area and who have rated movies. The data was collected into a Python collection object to plot the donut chart.

> Compared to the other figures in this book, the title of *Figure 8* is given in the middle. It is done using the `text()` method rather than using the `title()` method. This method can be used to print watermark text on the charts and plots.

Box plot

Frequently, it is a common requirement to compare the summary statistics of different datasets in one figure. The box plot is a very common plot used to capture the summary statistics of a dataset in an intuitive way. The following section does exactly the same, and to do this, *Figure 9* shows multiple box plots on a single figure.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> ages = spark.sql("SELECT occupation, age FROM users WHERE
occupation ='administrator' ORDER BY age")
>>> adminAges = ages.rdd.map(lambda p: p.age).collect()
>>> ages.describe().show()
  +-------+------------------+
  |summary|               age|
  +-------+------------------+
  |  count|                79|
  |   mean| 38.74683544303797|
  | stddev|11.052771408491363|
  |    min|                21|
  |    max|                70|
  +-------+------------------+
>>> ages = spark.sql("SELECT occupation, age FROM users WHERE
occupation ='engineer' ORDER BY age")
>>> engAges = ages.rdd.map(lambda p: p.age).collect()
>>> ages.describe().show()
  +-------+------------------+
  |summary|               age|
  +-------+------------------+
  |  count|                67|
  |   mean| 36.38805970149254|
  | stddev|11.115345348003853|
  |    min|                22|
  |    max|                70|
  +-------+------------------+
>>> ages = spark.sql("SELECT occupation, age FROM users WHERE
occupation ='programmer' ORDER BY age")
>>> progAges = ages.rdd.map(lambda p: p.age).collect()
>>> ages.describe().show()
  +-------+------------------+
  |summary|               age|
  +-------+------------------+
  |  count|                66|
  |   mean|33.121212121212125|
  | stddev| 9.551320948648684|
  |    min|                20|
  |    max|                63|
  +-------+------------------+

>>> # Box plots of the ages by profession
>>> boxPlotAges = [adminAges, engAges, progAges]
>>> boxPlotLabels = ['administrator','engineer', 'programmer' ]
>>> x = np.arange(len(boxPlotLabels))
>>> plt.figure()
>>> plt.boxplot(boxPlotAges)
```

```
>>> plt.title('Age summary statistics\n')
>>> plt.ylabel("Age")
>>> plt.xticks(x + 1, boxPlotLabels, rotation=0)
>>> plt.show(block=False)
```
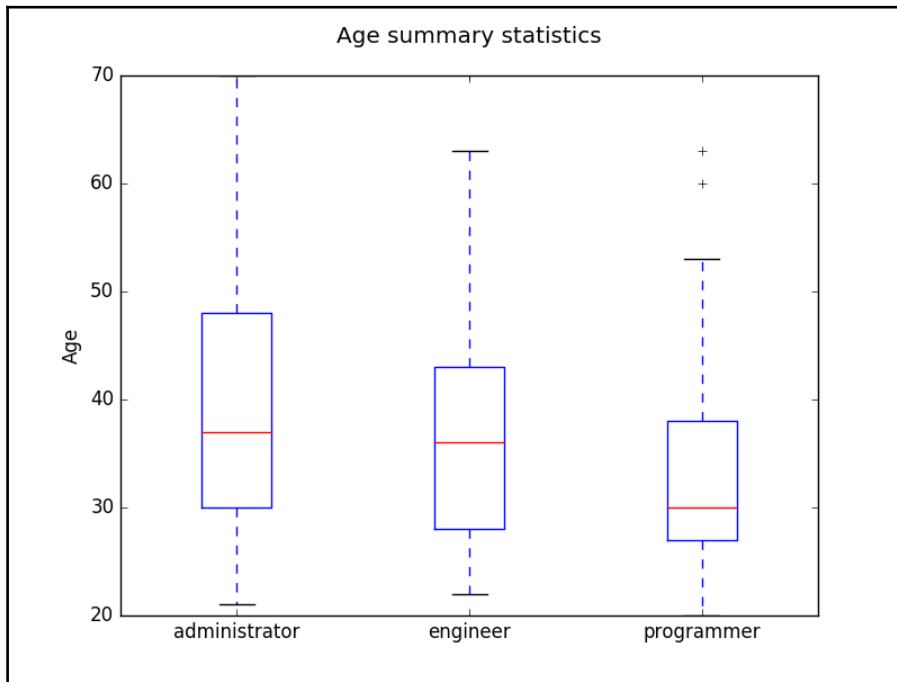

Figure 9

In the preceding section, a Spark DataFrame was created with occupation and age columns for each of three occupations: administrator, engineer, and programmer. Box plots were created for each of these datasets on one figure, which contains indicators for the minimum, 25th percentile, median, 75th percentile, maximum, and outlier values for each of the datasets to facilitate comparison. The box plot for the programmer occupation shows two value points represented by the + symbol. They are outlier values.

Vertical bar chart

In the preceding sections, the main dataset used for eliciting various charting and plotting use cases was the user data. The dataset that is taken up next is the movie dataset. In many datasets, to produce various charts and plots it will be a requirement to make the data suitable for the appropriate figure. Spark is rich with features to do the data processing.

The following use case demonstrates the preparation of data by applying some aggregation and using Spark SQL; the desired dataset is prepared for a classic bar chart containing the counts of movies by genre. *Figure 10* shows the bar chart after applying the aggregation operation on the movie data.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> movieLines = sc.textFile(dataDir + "u.item")
>>> splitMovieLines = movieLines.map(lambda l: l.split("|"))
>>> moviesRDD = splitMovieLines.map(lambda p: Row(id=p[0], title=p[1],
releaseDate=p[2], videoReleaseDate=p[3], url=p[4],
unknown=int(p[5]),action=int(p[6]),adventure=int(p[7]),animation=int(p[8]),
childrens=int(p[9]),comedy=int(p[10]),crime=int(p[11]),documentary=int(p[12
]),drama=int(p[13]),fantasy=int(p[14]),filmNoir=int(p[15]),horror=int(p[16]
),musical=int(p[17]),mystery=int(p[18]),romance=int(p[19]),sciFi=int(p[20])
,thriller=int(p[21]),war=int(p[22]),western=int(p[23])))
>>> moviesDF = spark.createDataFrame(moviesRDD)
>>> moviesDF.createOrReplaceTempView("movies")
>>> genreDF = spark.sql("SELECT sum(unknown) as unknown, sum(action) as
action,sum(adventure) as adventure,sum(animation) as animation,
sum(childrens) as childrens,sum(comedy) as comedy,sum(crime) as
crime,sum(documentary) as documentary,sum(drama) as drama,sum(fantasy) as
fantasy,sum(filmNoir) as filmNoir,sum(horror) as horror,sum(musical) as
musical,sum(mystery) as mystery,sum(romance) as romance,sum(sciFi) as
sciFi,sum(thriller) as thriller,sum(war) as war,sum(western) as western
FROM movies")
>>> genreList = genreDF.collect()
>>> genreDict = genreList[0].asDict()
>>> labelValues = list(genreDict.keys())
>>> countList = list(genreDict.values())
>>> genreDict
  {'animation': 42, 'adventure': 135, 'romance': 247, 'unknown': 2,
'musical': 56, 'western': 27, 'comedy': 505, 'drama': 725, 'war': 71,
'horror': 92, 'mystery': 61, 'fantasy': 22, 'childrens': 122, 'sciFi': 101,
'filmNoir': 24, 'action': 251, 'documentary': 50, 'crime': 109, 'thriller':
251}
>>> # Movie types and the counts
>>> x = np.arange(len(labelValues))
>>> plt.title('Movie types\n')
>>> plt.ylabel("Count")
>>> plt.bar(x, countList)
>>> plt.xticks(x + 0.5, labelValues, rotation=90)
>>> plt.gcf().subplots_adjust(bottom=0.20)
>>> plt.show(block=False)
```
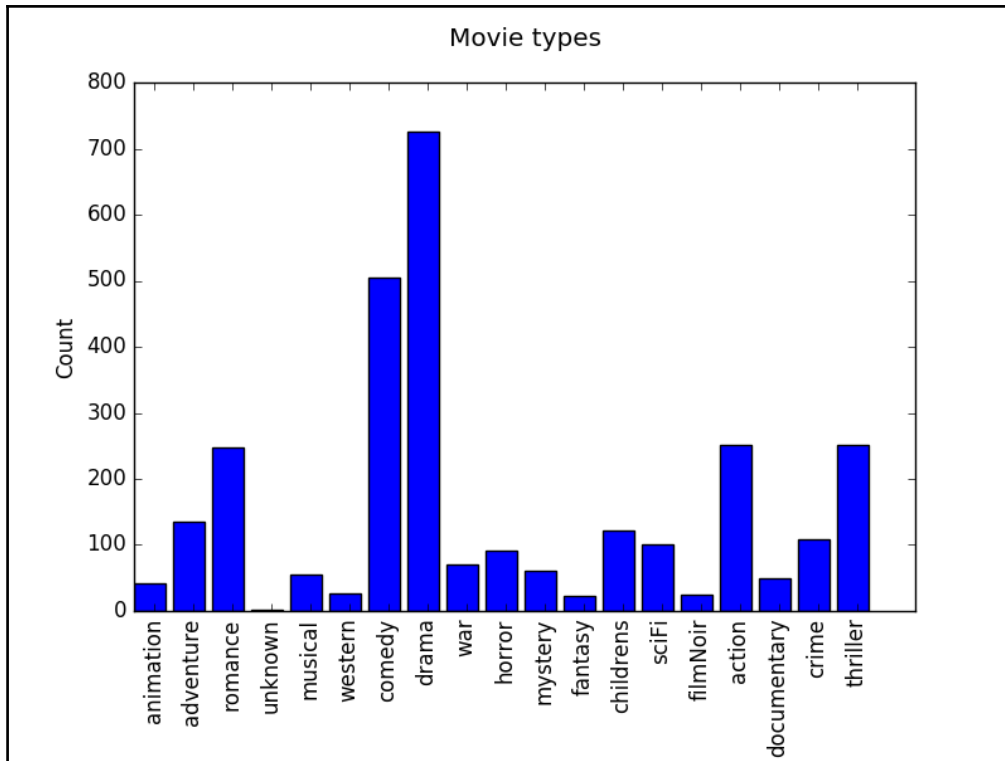
Figure 10

In the preceding section, a `SparkDataFrame` was created with the movie dataset. The genre of the movie was captured in separate columns. Across the dataset, an aggregation was done using Spark SQL, a new `SparkDataFrame` summary was created, and the data values were collected into a Python collection object. Since there are too many columns in the dataset, a Python function was used to convert that kind of data structure into a dictionary object containing the column name as the key and the selected single row value is the value of the key. From that dictionary, two datasets were created and a bar chart is drawn.

> When working with Spark, Python is used to develop data analysis applications, and it is almost certain that there are going to be many charts and plots. Instead of trying out all the code samples given in this chapter on the Python REPL for Spark, it is better to use IPython notebook as the IDE so that the code and the results can be seen together. The download section of this book contains the IPython notebook that contains all this code and the results. Readers can directly start using this.

Scatter plot

Scatter plots are very commonly used for plotting values that have two variables, such as a point in Cartesian space having an X value and Y value. In this movie dataset, the number of movies released in a given year shows this kind of behavior. In the scatter plots, typically, the values represented at the intersection points of the X coordinate and Y coordinate are points. Because of recent technology developments and the availability of sophisticated graphics packages, many use different shapes and colors to represent the points. In the following scatter plot, shown in *Figure 11*, small circles having a uniform area with random colors have been used to represent the values. When employing such intuitive and clever techniques to represent the points in scatter plots, care must be taken to make sure that it does not defeat the purpose and loses the simplicity that scatter plots offer to convey the behavior of the data. Simple and elegant shapes that do not clutter the Cartesian space are ideal for such non-point representations of the values.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> yearDF = spark.sql("SELECT substring(releaseDate,8,4) as
releaseYear, count(*) as movieCount FROM movies GROUP BY
substring(releaseDate,8,4) ORDER BY movieCount DESC LIMIT 10")
>>> yearDF.show()
    +-----------+----------+
    |releaseYear|movieCount|
    +-----------+----------+
    |       1996|       355|
    |       1997|       286|
    |       1995|       219|
    |       1994|       214|
    |       1993|       126|
    |       1998|        65|
    |       1992|        37|
    |       1990|        24|
    |       1991|        22|
    |       1986|        15|
    +-----------+----------+
>>> yearMovieCountTuple = yearDF.rdd.map(lambda p:
(int(p.releaseYear),p.movieCount)).collect()
>>> yearList,movieCountList = zip(*yearMovieCountTuple)
>>> countArea = yearDF.rdd.map(lambda p: np.pi *
(p.movieCount/15)**2).collect()
>>> plt.title('Top 10 movie release by year\n')
>>> plt.xlabel("Year")
>>> plt.ylabel("Number of movies released")
>>> plt.ylim([0,max(movieCountList) + 20])
>>> colors = np.random.rand(10)
>>> plt.scatter(yearList, movieCountList,c=colors)
```

```
>>> plt.show(block=False)
```
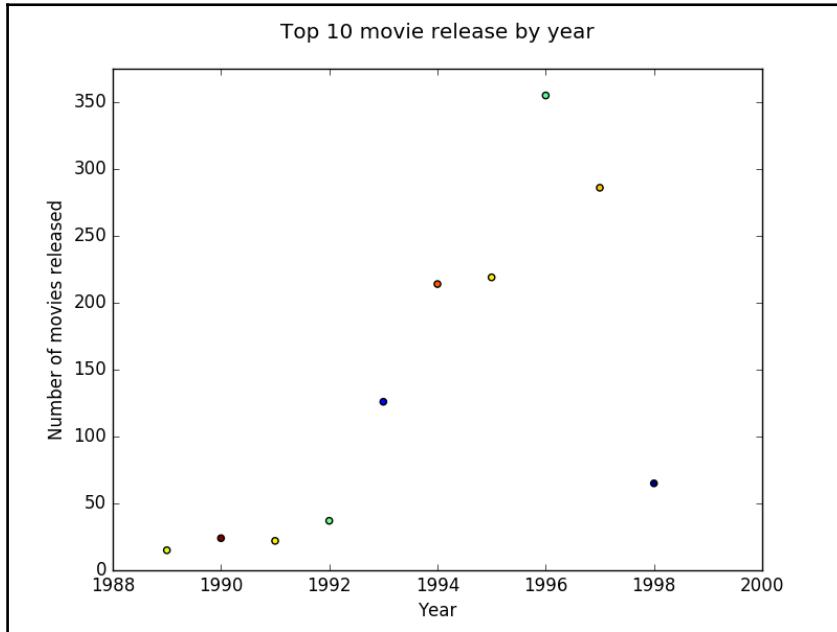

Figure 11

In the preceding section, a `SparkDataFrame` was used to collect the top 10 years in terms of the number of movies released in that year and the values were collected into a Python collection object and a scatter plot was drawn.

Enhanced scatter plot

Figure 11 is a very simple and elegant scatter plot, but it does not really convey the comparative behavior of a given plotted value as compared to the other values in the same space. For that, instead of representing the points as fixed-radius circles, if the points are drawn as circles with the area proportional to the value, that will give a different perspective. Figure 12 is going to show the scatter plot with the same data, but with circles that have a proportional area to represent the points.

As a continuation in the same Python REPL of Spark, run the following commands:

```
>>> # Top 10 years where the most number of movies have been released
>>> plt.title('Top 10 movie release by year\n')
>>> plt.xlabel("Year")
>>> plt.ylabel("Number of movies released")
>>> plt.ylim([0,max(movieCountList) + 100])
>>> colors = np.random.rand(10)
>>> plt.scatter(yearList, movieCountList,c=colors, s=countArea)
>>> plt.show(block=False)
```
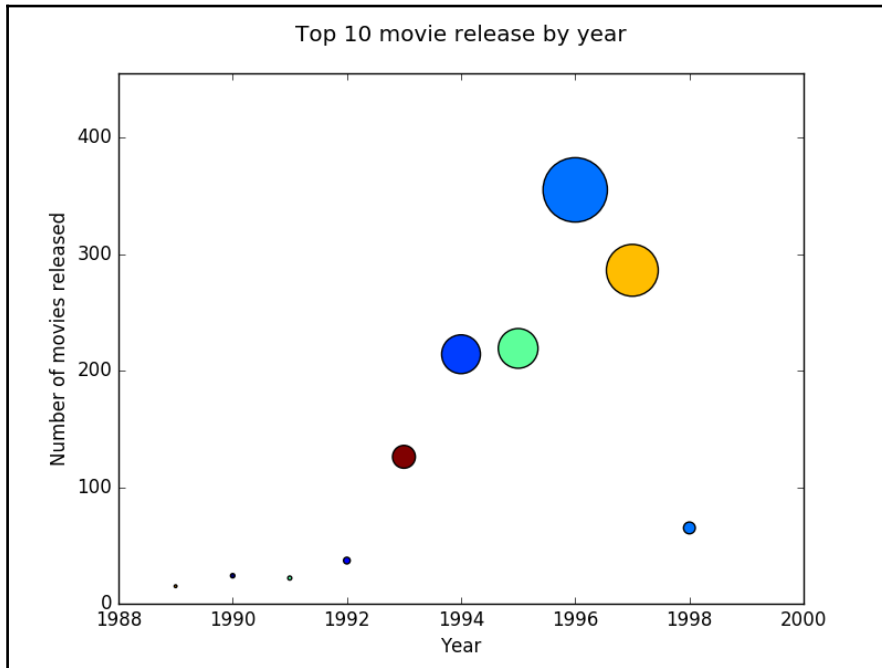

Figure 12

In the preceding section, the same dataset was used for *Figure 11* to draw the same scatter plot. Instead of plotting the points with uniform area circles, the points were drawn with proportionate area circles.

> **TIP**
> In all these code samples, the charts and plots are displayed using the show method. There are methods in matplotlib to save the generated charts and plots to disk, and they can be used for e-mailing, publishing to dashboards, and more.

Line graph

There are similarities between a scatter plot and a line graph. A scatter plot is ideal for representing individual data points, but taking all the points together gives a trend. A line graph also represents individual data points, but the points are connected. This is ideal for seeing the transition from one point to another point. In one figure, multiple line graphs can be drawn, enabling the comparison of two datasets. The preceding use case used a scatter plot to represent the number of movies released over a few years. Those numbers are just discrete data points plotted on one figure. If the need is to see a trend of how movie releases are changing over the years, a line graph is ideal. Similarly, if there is a need to compare movie releases over the years for two different genres, then one line can be used for each genre and both can be plotted on a single line graph. *Figure 13* is a line graph with multiple datasets.

As a continuation of the same Python REPL of Spark, run the following commands:

```
>>> yearActionDF = spark.sql("SELECT substring(releaseDate,8,4) as
actionReleaseYear, count(*) as actionMovieCount FROM movies WHERE action =
1 GROUP BY substring(releaseDate,8,4) ORDER BY actionReleaseYear DESC LIMIT
10")
>>> yearActionDF.show()
+-----------------+----------------+
|actionReleaseYear|actionMovieCount|
+-----------------+----------------+
|             1998|              12|
|             1997|              46|
|             1996|              44|
|             1995|              40|
|             1994|              30|
|             1993|              20|
|             1992|               8|
|             1991|               2|
|             1990|               7|
|             1989|               6|
+-----------------+----------------+
>>> yearActionDF.createOrReplaceTempView("action")
>>> yearDramaDF = spark.sql("SELECT substring(releaseDate,8,4) as
dramaReleaseYear, count(*) as dramaMovieCount FROM movies WHERE drama = 1
GROUP BY substring(releaseDate,8,4) ORDER BY dramaReleaseYear DESC LIMIT
10")
>>> yearDramaDF.show()
+----------------+---------------+
|dramaReleaseYear|dramaMovieCount|
+----------------+---------------+
|            1998|             33|
|            1997|            113|
```

```
         |          1996|           170|
         |          1995|            89|
         |          1994|            97|
         |          1993|            64|
         |          1992|            14|
         |          1991|            11|
         |          1990|            12|
         |          1989|             8|
         +--------------+--------------+
>>> yearDramaDF.createOrReplaceTempView("drama")
>>> yearCombinedDF = spark.sql("SELECT a.actionReleaseYear as
releaseYear, a.actionMovieCount, d.dramaMovieCount FROM action a, drama d
WHERE a.actionReleaseYear = d.dramaReleaseYear ORDER BY a.actionReleaseYear
DESC LIMIT 10")
>>> yearCombinedDF.show()
    +-----------+---------------+---------------+
    |releaseYear|actionMovieCount|dramaMovieCount|
    +-----------+---------------+---------------+
    |       1998|             12|             33|
    |       1997|             46|            113|
    |       1996|             44|            170|
    |       1995|             40|             89|
    |       1994|             30|             97|
    |       1993|             20|             64|
    |       1992|              8|             14|
    |       1991|              2|             11|
    |       1990|              7|             12|
    |       1989|              6|              8|
    +-----------+---------------+---------------+
>>> yearMovieCountTuple = yearCombinedDF.rdd.map(lambda p:
(p.releaseYear,p.actionMovieCount, p.dramaMovieCount)).collect()
>>> yearList,actionMovieCountList,dramaMovieCountList =
zip(*yearMovieCountTuple)
>>> plt.title("Movie release by year\n")
>>> plt.xlabel("Year")
>>> plt.ylabel("Movie count")
>>> line_action, = plt.plot(yearList, actionMovieCountList)
>>> line_drama, = plt.plot(yearList, dramaMovieCountList)
>>> plt.legend([line_action, line_drama], ['Action Movies', 'Drama
Movies'],loc='upper left')
>>> plt.gca().get_xaxis().get_major_formatter().set_useOffset(False)
>>> plt.show(block=False)
```
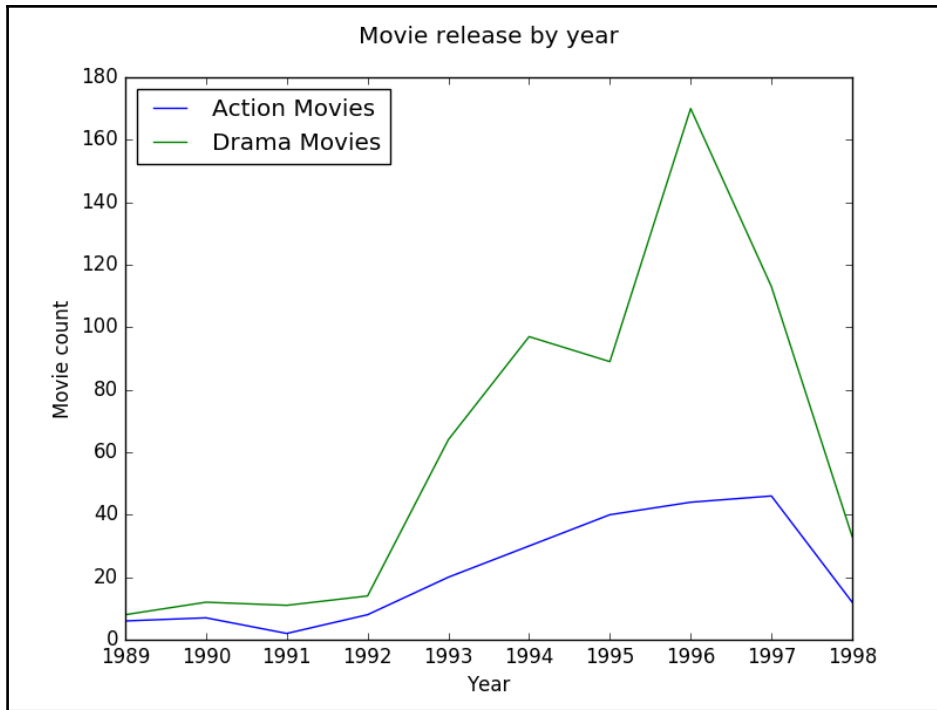
Figure 13

In the preceding section, Spark DataFrames were created to get the datasets for the number of action movies and drama movies released over the period of the last 10 years. The data was collected into Python collection objects and line graphs were drawn in the same figure.

Python, in conjunction with the matplotlib library, is very rich in terms of methods to produce publication-quality charts and plots. Spark can be used as the workhorse for processing the data coming from heterogeneous sources of data, and the results can also be saved to a wide variety of data formats.

Those who are exposed to the Python data analysis library **pandas** will find it easy to understand the material covered in this chapter because Spark DataFrames designed from the ground up by taking inspiration from the R DataFrame as well as **pandas**.

This chapter has covered only a few sample charts and plots that can be created using the **matplotlib** library. The main idea of this chapter was to help the reader understand the capability of using this library in conjunction with Spark, where Spark is doing the data processing, and **matplotlib** is doing the charting and plotting.

The data file used in this chapter is read from a local filesystem. Instead of this, it can be read from HDFS or any other Spark-supported data source.

When using Spark as the primary framework for data processing, the most important point to keep in mind is that any possible data processing is to be done by Spark, mainly because Spark can do data processing in the best way. Only the processed data is to be returned to the Spark driver program for doing the charting and plotting.

References

For more information please refer to following links:

- `http://www.numpy.org/`
- `http://www.scipy.org/`
- `http://matplotlib.org/`
- `https://movielens.org/`
- `http://grouplens.org/datasets/movielens/`
- `http://pandas.pydata.org/`

Summary

Processed data is used for data analysis. Data analysis requires a deep understanding of the processed data. Charts and plots enhance the understanding of the characteristics of the underlying data. In essence, for a data analysis application, data processing, charting, and plotting are essential. This chapter has covered the usage of Spark with Python, in conjunction with Python charting and plotting libraries, for developing data analysis applications.

In most organizations, business requirements are driving the need to build data processing applications involving the real-time ingestion of data, in various shapes and forms, with tremendous velocity. This mandates the need to process a stream of incoming data to the organizational data sink. The next chapter is going to discuss Spark Streaming, which is a library that works on top of Spark and enables the processing of various types of data streams.

6
Spark Stream Processing

Data processing use cases can be mainly divided into two types. The first type is the use cases where the data is static and processing is done in its entirety as one unit of work, or by dividing it into smaller batches. While doing the data processing, the underlying data set does not change nor do new data sets get added to the processing units. This is batch processing.

The second type is the use cases where the data is getting generated like a stream, and the processing is done as and when the data is generated. This is stream processing. In the previous chapters of this book, all the data processing use cases were pertaining to the former type. This chapter is going to focus on the latter.

We will cover the following topics in this chapter:

- Data stream processing
- Micro batch data processing
- A log event processor
- Windowed data processing and other options
- Kafka stream processing
- Streaming jobs with Spark

Data stream processing

Data sources generate data like a stream, and many real-world use cases require them to be processed in real time. The meaning of *real time* can change from use case to use case. The main parameter that defines what is meant by real time for a given use case is how soon the ingested data or the frequent interval in which all the data ingested since the last interval needs to be processed. For example, when a major sports event is happening, the application that consumes the score events and sends them to the subscribed users should be processing the data as fast as it can. The faster can be sent, the better it is.

But what is the definition of *fast* here? Is it fine to process the score data within, say, an hour of the score event happening? Probably not. Is it fine to process the data within a minute of the score event happening? It is definitely better than processing after an hour. Is it fine to process the data within a second of the score event happening? Probably yes, and much better than the earlier data processing time intervals.

In any data stream processing use cases, this time interval is very important. The data processing framework should have the capability to process the data stream in an appropriate time interval of choice to deliver good business value.

When processing stream data in regular intervals of choice, the data is collected from the beginning of the time interval to the end of the time interval, grouped in a micro batch, and data processing is done on that batch of data. Over an extended period of time, the data processing application would have processed many such micro batches of data. In this type of processing, the data processing application will have visibility of only the specific micro batch that is getting processed at a given point in time. In other words, the application will not have any visibility or access to the already processed micro batches of data.

Now, there is another dimension to this type of processing. Suppose a given use case mandates the need to process the data every minute, but at the same time, while processing the data of a given micro batch, there is a need to peek into the data that was already processed in the last 15 minutes. A fraud detection module of a retail banking transaction processing application is a good example of this particular business requirement. There is no doubt that the retail banking transactions are to be processed within milliseconds of their occurrence. When processing an ATM cash withdrawal transaction, it is a good idea to see whether somebody is trying to continuously withdraw cash and, if found, send the proper alert. For this, when processing a given cash withdrawal transaction, the application checks whether there are any other cash withdrawals from the same ATM using the same card that was used in the last 15 minutes. The business rule is to send an alert when such transactions are more than two in the last 15 minutes. In this use case, the fraud detection application should have visibility of all the transactions that happened in a window of 15 minutes.

A good stream data processing framework should have the ability to processing the data in any given interval of time, as well as the ability to peek into the data ingested within a sliding window of time. The Spark Streaming library that is working on top of Spark is one of the best data stream processing frameworks that has both of these capabilities.

Look again at the bigger picture of the Spark library stack as given in *Figure 1* to set the context and see what is being discussed here before getting into and taking up the use cases.
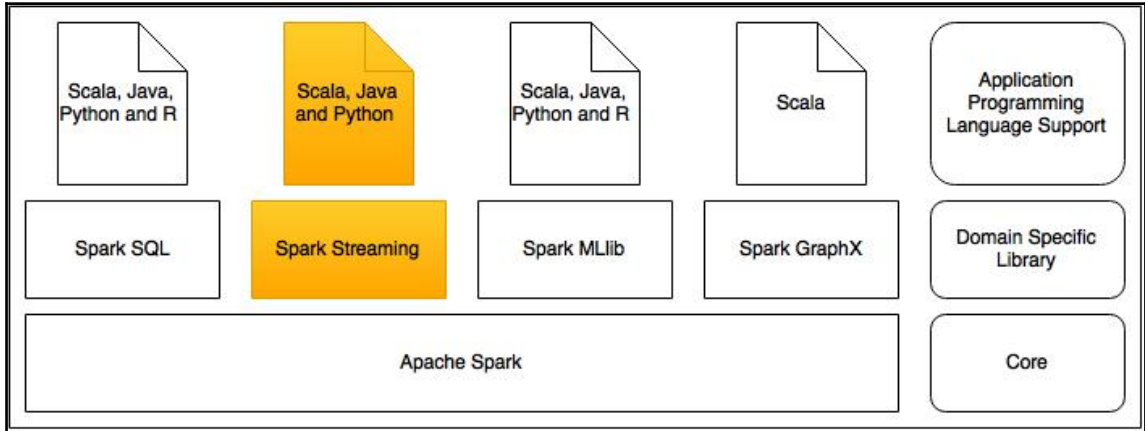
Figure 1

Micro batch data processing

Every Spark Streaming data processing application will be running continuously till it is terminated. This application will be constantly *listening* to the data source to receive the incoming stream of data. The Spark Streaming data processing application would have a configured batch interval. At the end of every batch interval, it will produce a data abstraction named **Discretized Stream** (**DStream**) which works very similar to Spark's RDD. Just like RDD, a DStream supports an equivalents method for the commonly used Spark transformations and Spark actions.

Just like RDD, a DStream is also immutable and distributed.

Figure 2 shows how DStreams are being produced in a Spark Streaming data processing application.
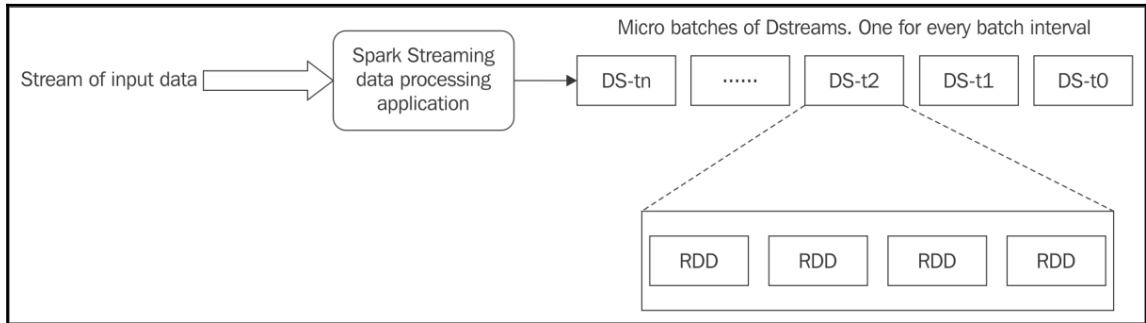
Figure 2

Figure 2 depicts the most important elements of a Spark Streaming application. For the configured batch interval, the application produces one DStream. Each DStream is a collection of RDDs consisting of the data collected within that batch interval. The number of RDDs within a DStream for a given batch interval will vary.

> **TIP**
>
> Since Spark Streaming applications are continuously running applications collecting data, in this chapter, rather than running the code in REPL, the complete application is discussed, including the instructions to compile, package and run.

The Spark programming model was discussed in `Chapter 2`, *Spark Programming Model*.

Programming with DStreams

Programming with DStreams in a Spark Streaming data processing application also follows a very similar model, as DStreams consist of one or more RDDs. When methods such as Spark transformations or Spark actions are invoked on a DStream, the equivalent operation is applied to all the RDDs that constitute the DStream.

> An important point to note here is that not all the Spark transformations and Spark actions that work on RDD are unsupported on DStreams. The other notable change is the differences in capability across programming languages.
>
> The Scala and Java APIs for Spark Streaming are ahead of the Python API in terms of the number of features supported for Spark Streaming data processing application development.

Figure 3 depicts how methods applied on a DStream are applied on the underlying RDDs. The Spark Streaming programming guide is to be consulted before using any of the methods on DStreams. The Spark Streaming programming guide is marked with special callouts containing the text *Python API* wherever the Python API deviates from its Scala or Java counterparts.

Assume that, for a given batch interval in a Spark Streaming data processing application, a DStream is generated consisting of multiple RDDs. When a filter method is applied on that DStream, here is how it gets translated into the underlying RDDs. *Figure 3* shows a filter transformation applied on a DStream with two RDDs, resulting in another DStream containing only one RDD because of the filter condition:
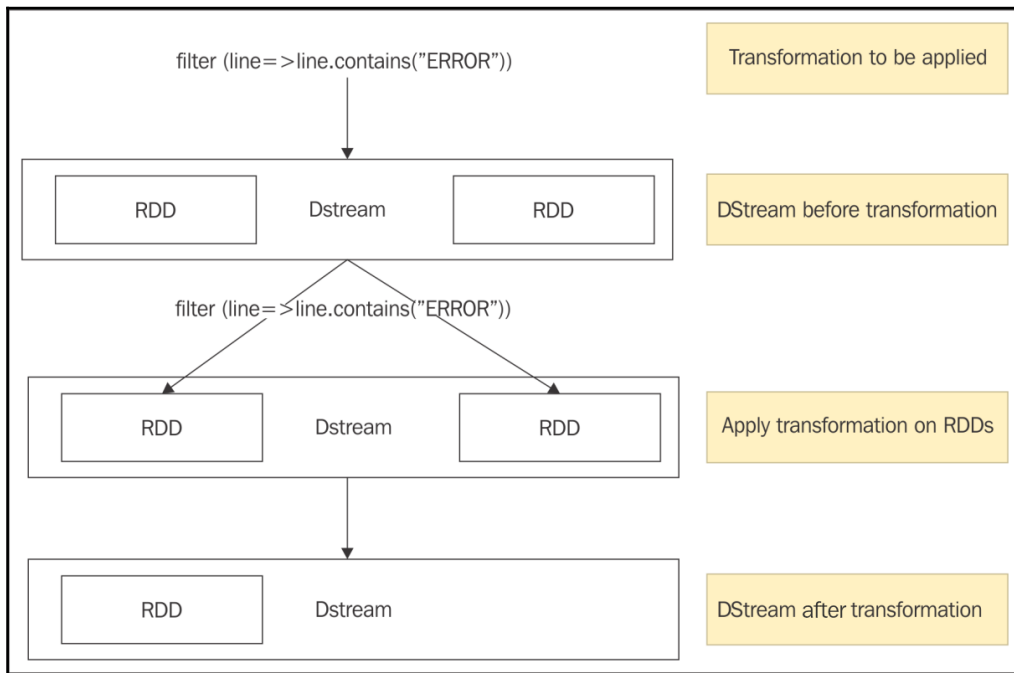
Figure 3

A log event processor

These days, it is very common to have a central repository of application log events in many enterprises. Also, the log events are streamed live to data processing applications in order to monitor the performance of the running applications on a real-time basis so that timely remediation measures can be taken. Such a use case is discussed here to demonstrate the real-time processing of log events using a Spark Streaming data processing application. In this use case, the live application log events are written to a TCP socket. The Spark Streaming data processing application constantly listens to a given port on a given host to collect the stream of log events.

Getting ready with the Netcat server

The Netcat utility that comes with most UNIX installations is used here as the data server. To make sure that Netcat is installed in the system, type the manual command as given in the following scripts, and, after coming out of it, run it and make sure that there is no error message. Once the server is up and running, whatever is typed in the standard input of the Netcat server console is considered as the application logs events for simplicity and demonstration purposes. The following commands run from a terminal prompt will start the Netcat data server on localhost port `9999`:

```
$ man nc
NC(1)               BSD General Commands Manual
NC(1)
NAME
     nc -- arbitrary TCP and UDP connections and listens
SYNOPSIS
     nc [-46AcDCdhklnrtUuvz] [-b boundif] [-i interval] [-p source_port] [-
s source_ip_address] [-w timeout] [-X proxy_protocol] [-x
proxy_address[:port]]
         [hostname] [port[s]]
 DESCRIPTION
     The nc (or netcat) utility is used for just about anything under the
sun involving TCP or UDP.  It can open TCP connections, send UDP packets,
listen on
     arbitrary TCP and UDP ports, do port scanning, and deal with both IPv4
and IPv6.  Unlike telnet(1), nc scripts nicely, and separates error
messages onto
     standard error instead of sending them to standard output, as
telnet(1) does with some.
     Common uses include:
         o    simple TCP proxies
         o    shell-script based HTTP clients and servers
         o    network daemon testing
```

```
            o    a SOCKS or HTTP ProxyCommand for ssh(1)
            o    and much, much more
$ nc -lk 9999
```

Once the preceding steps are completed, the Netcat server is ready and the Spark Streaming data processing application will process all the lines that are typed in the previous console window. Leave this console window alone; all the following shell commands will be run in a different terminal window.

Since there is a lack of parity of Spark Streaming features between different programming languages, the Scala code is used to explain all the Spark Streaming concepts and use cases. After that, the Python code is given, and, if there is a lack of support for any of the features being discussed in Python, that is also captured.

The Scala and Python code are organized in the way demonstrated in *Figure 4*. For the compilation, packaging and running of the code, bash scripts are used so that it is easy for the readers to run them to produce consistent results. Each of these script file contents are discussed here.

Organizing files

In the following folder tree, the `project` and `target` folders are created at runtime. The source code that comes with this book can be copied directly to a convenient folder in the system:
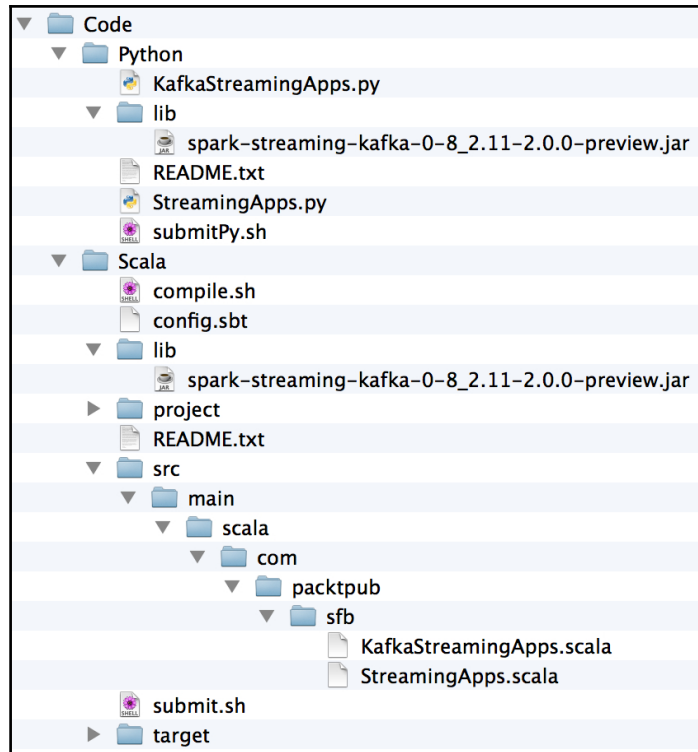
Figure 4

For compiling and packaging, the **Scala build tool** (**sbt**) is used. In order to make sure that sbt is working properly, run the following commands from the Scala folder of the tree in *Figure 4* in the terminal window. This is to make sure that sbt is working fine and the code is compiling:

```
$ cd Scala
$ sbt
> compile
  [success] Total time: 1 s, completed 24 Jul, 2016 8:39:04 AM
> exit
$
```

The following table captures the representative sample list of files and the purpose of each of them in the context of the Spark Streaming data processing application that is being discussed here.

File Name	Purpose
README.txt	Instructions to run the application. One for the Scala application and the other one for the Python application.
submitPy.sh	Bash script to submit the Python job to the Spark cluster.
compile.sh	Bash script to compile the Scala code.
submit.sh	Bash script to submit the Scala job to the Spark cluster.
config.sbt	The sbt configuration file.
*.scala	Spark Streaming data processing application code in Scala.
*.py	Spark Streaming data processing application code in Python.
*.jar	The Spark Streaming and Kafka integration JAR file that needs to be downloaded and placed under the lib folder for the proper functioning of the applications. This is being used in submit.sh as well as in submitPy.sh for submitting the job to the cluster.

Submitting the jobs to the Spark cluster

To properly run the application, some of the configurations depend on the system in which it is being run. They are to be edited in the submit.sh file and the submitPy.sh file. Wherever such edits are required, comments are given with the [FILLUP] tag. The most important of these are the setting of the Spark installation directory and the Spark master configuration, which can differ from system to system. The source of the preceding script submit.sh file is given as follows:

```bash
#!/bin/bash
#-----------
# submit.sh
#-----------
# IMPORTANT – Assumption is that the $SPARK_HOME and $KAFKA_HOME
environment variables are already set in the system that is running the
application
# [FILLUP] Which is your Spark master. If monitoring is needed, use the
desired Spark master or use local
# When using the local mode. It is important to give more than one
cores in square brackets
#SPARK_MASTER=spark://Rajanarayanans-MacBook-Pro.local:7077
```

```
    SPARK_MASTER=local[4]
    # [OPTIONAL] Your Scala version
    SCALA_VERSION="2.11"
    # [OPTIONAL] Name of the application jar file. You should be OK to
leave it like that
    APP_JAR="spark-for-beginners_$SCALA_VERSION-1.0.jar"
    # [OPTIONAL] Absolute path to the application jar file
    PATH_TO_APP_JAR="target/scala-$SCALA_VERSION/$APP_JAR"
    # [OPTIONAL] Spark submit command
    SPARK_SUBMIT="$SPARK_HOME/bin/spark-submit"
    # [OPTIONAL] Pass the application name to run as the parameter to this
script
    APP_TO_RUN=$1
    sbt package
    if [ $2 -eq 1 ]
    then
    $SPARK_SUBMIT --class $APP_TO_RUN --master $SPARK_MASTER --jars
$KAFKA_HOME/libs/kafka-
clients-0.8.2.2.jar,$KAFKA_HOME/libs/kafka_2.11-0.8.2.2.jar,$KAFKA_HOME/lib
s/metrics-core-2.2.0.jar,$KAFKA_HOME/libs/zkclient-0.3.jar,./lib/spark-
streaming-kafka-0-8_2.11-2.0.0-preview.jar $PATH_TO_APP_JAR
    else
    $SPARK_SUBMIT --class $APP_TO_RUN --master $SPARK_MASTER --jars
$PATH_TO_APP_JAR $PATH_TO_APP_JAR
    fi
```

The source of the preceding script file `submitPy.sh` is given as follows:

```
    #!/usr/bin/env bash
    #------------
    # submitPy.sh
    #------------
    # IMPORTANT - Assumption is that the $SPARK_HOME and $KAFKA_HOME
environment variables are already set in the system that is running the
application
    # Disable randomized hash in Python 3.3+ (for string) Otherwise the
following exception will occur
    # raise Exception("Randomness of hash of string should be disabled via
PYTHONHASHSEED")
    # Exception: Randomness of hash of string should be disabled via
PYTHONHASHSEED
    export PYTHONHASHSEED=0
    # [FILLUP] Which is your Spark master. If monitoring is needed, use the
desired Spark master or use local
    # When using the local mode. It is important to give more than one
cores in square brackets
    #SPARK_MASTER=spark://Rajanarayanans-MacBook-Pro.local:7077
    SPARK_MASTER=local[4]
```

```
    # [OPTIONAL] Pass the application name to run as the parameter to this
script
    APP_TO_RUN=$1
    # [OPTIONAL] Spark submit command
    SPARK_SUBMIT="$SPARK_HOME/bin/spark-submit"
    if [ $2 -eq 1 ]
    then
    $SPARK_SUBMIT --master $SPARK_MASTER --jars $KAFKA_HOME/libs/kafka-
clients-0.8.2.2.jar,$KAFKA_HOME/libs/kafka_2.11-0.8.2.2.jar,$KAFKA_HOME/lib
s/metrics-core-2.2.0.jar,$KAFKA_HOME/libs/zkclient-0.3.jar,./lib/spark-
streaming-kafka-0-8_2.11-2.0.0-preview.jar $APP_TO_RUN
    else
    $SPARK_SUBMIT --master $SPARK_MASTER $APP_TO_RUN
    fi
```

Monitoring running applications

As described in `Chapter 2`, *Spark Programming Model*, Spark installation comes with a powerful Spark web UI for monitoring the Spark applications that are running.

There are additional visualizations available specifically for the Spark Streaming jobs that are running.

The following scripts start the Spark master and workers, and enable monitoring. The assumption here is that the reader has made all the configuration changes suggested in `Chapter 2`, *Spark Programming Model* to enable Spark application monitoring. If that is not done, the applications can still be run. The only change to be made is to put the cases in the `submit.sh` file and the `submitPy.sh` file to make sure that instead of the Spark master URL, something like `local[4]` is used. Run the following commands on the terminal window:

```
    $ cd $SPARK_HOME
    $ ./sbin/start-all.sh
      starting org.apache.spark.deploy.master.Master, logging to
/Users/RajT/source-code/spark-source/spark-2.0/logs/spark-RajT-
org.apache.spark.deploy.master.Master-1-Rajanarayanans-MacBook-
Pro.local.out
      localhost: starting org.apache.spark.deploy.worker.Worker, logging to
/Users/RajT/source-code/spark-source/spark-2.0/logs/spark-RajT-
org.apache.spark.deploy.worker.Worker-1-Rajanarayanans-MacBook-
Pro.local.out
```

Make sure that the Spark web UI is up and running by visiting
`http://localhost:8080/`.

Implementing the application in Scala

The following code snippet is the Scala code for the log event processing application:

```scala
/**
The following program can be compiled and run using SBT
Wrapper scripts have been provided with this
The following script can be run to compile the code
./compile.sh
The following script can be used to run this application in Spark
./submit.sh com.packtpub.sfb.StreamingApps
**/
package com.packtpub.sfb
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.storage.StorageLevel
import org.apache.log4j.{Level, Logger}
object StreamingApps{
  def main(args: Array[String])
  {
   // Log level settings
   LogSettings.setLogLevels()
   // Create the Spark Session and the spark context
   val spark = SparkSession
         .builder
         .appName(getClass.getSimpleName)
         .getOrCreate()
   // Get the Spark context from the Spark session for creating the
   streaming context
   val sc = spark.sparkContext
   // Create the streaming context
   val ssc = new StreamingContext(sc, Seconds(10))
   // Set the check point directory for saving the data to recover when
   there is a crash
   ssc.checkpoint("/tmp")
   println("Stream processing logic start")
   // Create a DStream that connects to localhost on port 9999
   // The StorageLevel.MEMORY_AND_DISK_SER indicates that the data will
be
   stored in memory and if it overflows, in disk as well
   val appLogLines = ssc.socketTextStream("localhost", 9999,
   StorageLevel.MEMORY_AND_DISK_SER)
   // Count each log message line containing the word ERROR
   val errorLines = appLogLines.filter(line => line.contains("ERROR"))
   // Print the elements of each RDD generated in this DStream to the
    console
   errorLines.print()
   // Count the number of messages by the windows and print them
```

```
      errorLines.countByWindow(Seconds(30), Seconds(10)).print()
      println("Stream processing logic end")
      // Start the streaming
      ssc.start()
      // Wait till the application is terminated
      ssc.awaitTermination()
    }
  }
  object LogSettings{
    /**
     Necessary log4j logging level settings are done
    */
    def setLogLevels() {
      val log4jInitialized =
       Logger.getRootLogger.getAllAppenders.hasMoreElements
      if (!log4jInitialized) {
         // This is to make sure that the console is clean from other
INFO
          messages printed by Spark
         Logger.getRootLogger.setLevel(Level.WARN)
      }
    }
  }
```

In the previous code snippet, there are two Scala objects. One is for setting the proper logging levels, to make sure that unwanted messages are not displayed on the console. The StreamingApps Scala object holds the logic of the stream processing. The following list captures the essence of the functionality:

- A Spark configuration is created with the application name.
- A Spark StreamingContext object is created, which is the heart of the stream processing. The second parameter of the StreamingContext constructor is the batch interval, which is 10 seconds. The line containing ssc.socketTextStream creates DStreams at every batch interval, which is 10 seconds here, containing the lines typed in the Netcat console.
- A filter transformation is applied next on the DStream, to have only the lines containing the word ERROR. The filter transformation creates new DStreams containing only the filtered lines.
- The next line prints the DStream contents to the console. In other words, for every batch interval, if there are lines containing the word ERROR, that get displayed in the console.
- At the end of this data processing logic, the given StreamingContext is started and will run until it is terminated.

In the previous code snippet, there is no loop construct telling the application to repeat till the running application is terminated. This is achieved by the Spark Streaming library itself. From the beginning till the termination of the data processing application, all the statements are run once. All the operations on the DStreams are repeated (internally) for every batch. If the output of the previous application is closely examined, the output from the println() statements are seen only once in the console, even though these statements are between the initialization and termination of the StreamingContext. That is because the *magic loop* is repeating only for the statements containing original and derived DStreams.

Because of the special nature of the looping implemented in Spark Streaming applications, it is futile to give print statements and log statements within the streaming logic in the application code, like the one that is given in the code snippet. If that is a must, then these logging statements are to be instrumented within the functions that are passed to DStreams for the transformations and actions.

> If persistence is required for the processed data, there are many output operations available for DStreams, just as there are for RDDs.

Compiling and running the application

The following commands are run on the terminal window to compile and run the application. Instead of using ./compile.sh, a simple sbt compile command can also be used.

> Note that, as discussed previously, the Netcat server must be running before these commands are executed.

```
$ cd Scala
$ ./compile.sh
  [success] Total time: 1 s, completed 24 Jan, 2016 2:34:48 PM
$ ./submit.sh com.packtpub.sfb.StreamingApps
  Stream processing logic start
  Stream processing logic end
  -------------------------------------------
  Time: 1469282910000 ms
  -------------------------------------------
  -------------------------------------------
  Time: 1469282920000 ms
  -------------------------------------------
```

If no error messages are shown, and the results are showing in line with the previous output, the Spark Streaming data processing application has started properly.

Handling the output

Note that the output of the print statements comes before the DStream output print. So far, nothing has been typed in the Netcat console and, therefore, there is nothing to process.

Now go to the Netcat console that was started earlier and enter the following lines of log event messages by giving a gap of few seconds to make sure that the output goes to more than one batch, where the batch size is 10 seconds:

```
[Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:46:23 2015] [WARN] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /apache/web/test
[Fri Dec 20 01:54:34 2015] [WARN] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /apache/web/test
[Fri Dec 20 02:25:55 2015] [ERROR] [client 1.2.3.4.5.6] Client sent
malformed Host header
[Fri Dec 20 02:25:55 2015] [WARN] [client 1.2.3.4.5.6] Client sent
malformed Host header
[Mon Dec 20 23:02:01 2015] [ERROR] [client 1.2.3.4.5.6] user test:
authentication failure for "/~raj/test": Password Mismatch
[Mon Dec 20 23:02:01 2015] [WARN] [client 1.2.3.4.5.6] user test:
authentication failure for "/~raj/test": Password Mismatch
```

Once the log event messages are entered into the Netcat console window, the following results will start showing up in the Spark Streaming data processing application, filtering only the log event messages containing the keyword ERROR.

```
        -------------------------------------------
        Time: 1469283110000 ms
        -------------------------------------------
        [Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory
index
        forbidden by rule: /home/raj/
        -------------------------------------------
        Time: 1469283190000 ms
        -------------------------------------------
        -------------------------------------------

        Time: 1469283200000 ms
```

```
        --------------------------------------------

        [Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory
  index
        forbidden by rule: /apache/web/test
        --------------------------------------------
        Time: 1469283250000 ms
        --------------------------------------------


        --------------------------------------------
        Time: 1469283260000 ms
        --------------------------------------------
        [Fri Dec 20 02:25:55 2015] [ERROR] [client 1.2.3.4.5.6] Client sent
        malformed Host header
        --------------------------------------------

        Time: 1469283310000 ms
        --------------------------------------------
        [Mon Dec 20 23:02:01 2015] [ERROR] [client 1.2.3.4.5.6] user test:
        authentication failure for "/~raj/test": Password Mismatch
        --------------------------------------------
        Time: 1453646710000 ms
        --------------------------------------------
```

The Spark web UI (`http://localhost:8080/`) was already enabled, and Figures 5 and 6 show the Spark applications and statistics.

From the main page (after visiting the URL `http://localhost:8080/`), click the running Spark Streaming data processing application's name link to bring up the regular monitoring page. From that page, click the **Streaming** tab, to reveal the page containing the streaming statistics.

The link and tab to be clicked are circled in red:
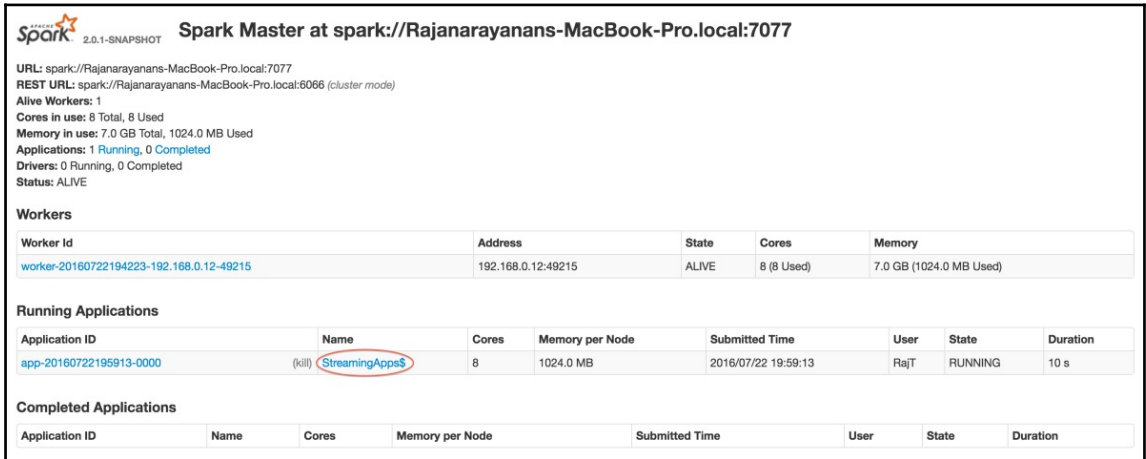
Figure 5

From the page shown in *Figure 5*, click on the circled application link; it will take you to the relevant page. From that page, once the **Streaming** tab is clicked, the page containing the streaming statistics will show up as captured in *Figure 6*:
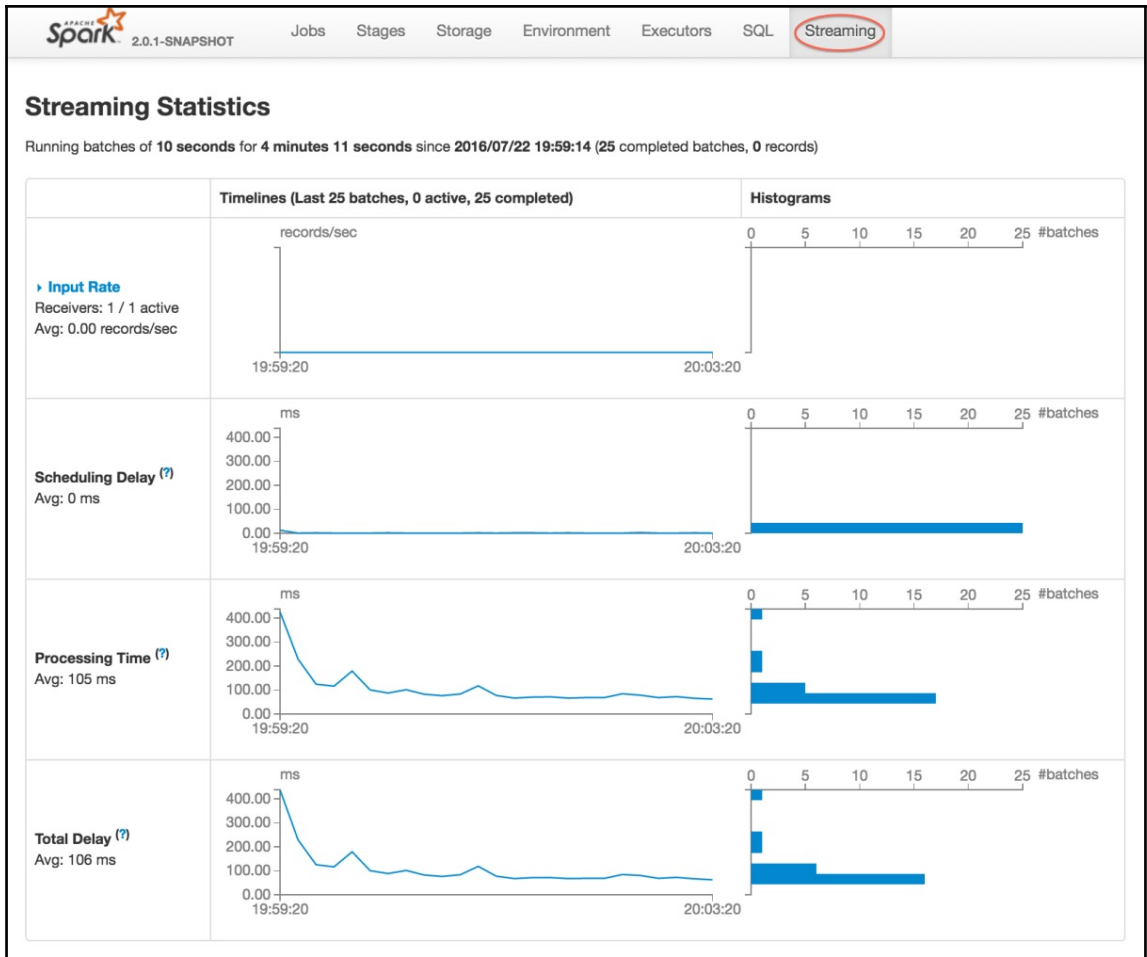
Figure 6

There are a whole lot of application statistics available from these Spark web UI pages, and exploring them extensively is a good idea to gain a deeper understanding of the behavior of the Spark Streaming data processing applications submitted.

> **TIP**
>
> Care must be taken while enabling the monitoring of streaming applications as it should not affect the performance of the application itself.

Implementing the application in Python

The same use case is implemented in Python, and the following code snippet saved in `StreamingApps.py` is used to do this:

```python
# The following script can be used to run this application in Spark
# ./submitPy.sh StreamingApps.py
from __future__ import print_function
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
if __name__ == "__main__":
    # Create the Spark context
    sc = SparkContext(appName="PythonStreamingApp")
    # Necessary log4j logging level settings are done
    log4j = sc._jvm.org.apache.log4j
    log4j.LogManager.getRootLogger().setLevel(log4j.Level.WARN)
    # Create the Spark Streaming Context with 10 seconds batch interval
    ssc = StreamingContext(sc, 10)
    # Set the check point directory for saving the data to recover when
    there is a crash
    ssc.checkpoint("\tmp")
    # Create a DStream that connects to localhost on port 9999
    appLogLines = ssc.socketTextStream("localhost", 9999)
    # Count each log messge line containing the word ERROR
    errorLines = appLogLines.filter(lambda appLogLine: "ERROR" in
    appLogLine)
    # // Print the elements of each RDD generated in this DStream to
the
    console
    errorLines.pprint()
    # Count the number of messages by the windows and print them
    errorLines.countByWindow(30,10).pprint()
    # Start the streaming
    ssc.start()
    # Wait till the application is terminated
    ssc.awaitTermination()
```

The following commands are run on the terminal window to run the Python Spark Streaming data processing application from the directory where the code is downloaded. Before running the application, in the same way that the modifications are made to the script that is used to run the Scala application, the `submitPy.sh` file also has to be changed to point to the right Spark installation directory and configure the Spark master. If monitoring is enabled, and if the submission is pointing to the right Spark master, the same Spark web UI will capture the statistics of the Python Spark Streaming data processing applications as well.

The following commands are run on the terminal window to run the Python application:

```
$ cd Python
$ ./submitPy.sh StreamingApps.py
```

Once the same log event messages used in the Scala implementation are entered into the Netcat console window, the following results will start showing up in the streaming application, filtering only the log event messages containing the keyword ERROR:

```
-------------------------------------------
Time: 2016-07-23 15:21:50
-------------------------------------------
-------------------------------------------
Time: 2016-07-23 15:22:00
-------------------------------------------
[Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory
index
forbidden by rule: /home/raj/
-------------------------------------------
Time: 2016-07-23 15:23:50
-------------------------------------------
[Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory
index
forbidden by rule: /apache/web/test
-------------------------------------------
Time: 2016-07-23 15:25:10
-------------------------------------------
-------------------------------------------
Time: 2016-07-23 15:25:20
-------------------------------------------
[Fri Dec 20 02:25:55 2015] [ERROR] [client 1.2.3.4.5.6] Client sent
malformed Host header
-------------------------------------------
Time: 2016-07-23 15:26:50
-------------------------------------------
[Mon Dec 20 23:02:01 2015] [ERROR] [client 1.2.3.4.5.6] user test:
authentication failure for "/~raj/test": Password Mismatch
-------------------------------------------
Time: 2016-07-23 15:26:50
-------------------------------------------
```

If you look at the outputs from both the Scala and Python programs, you can clearly see whether there are any log event messages containing the word ERROR in a given batch interval. Once the data is processed, the application discards the processed data without retaining them for any future use.

In other words, the application never retains or remembers any of the log event messages from the previous batch intervals. If there is a need to capture the number of error messages, say in the last 5 minutes or so, then the previous approach will not work. We will discuss that in the next section.

Windowed data processing

In the Spark Streaming data processing application discussed in the previous section, assume that there is a need to count the number of log event messages containing the keyword ERROR in the previous three batches. In other words, there should be the ability to count the number of such event messages across a window of three batches. At any given point in time, the window should be sliding along with time as and when a new batch of data is available. Three important terms have been discussed here, and *Figure 7* explains them. They are:

- Batch interval: The time interval at which a DStream is produced
- Window length: The duration of the number of batch intervals where there is a need to peek into all the DStreams produced in those batch intervals
- Sliding interval: The interval at which the window operation, such as counting the event messages, is performed
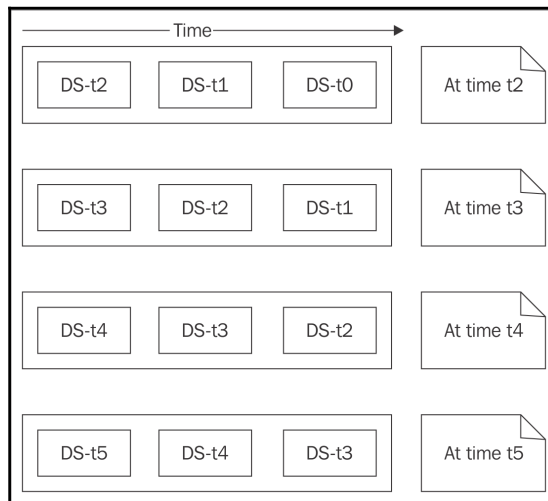
Figure 7

In *Figure 7*, at a given point in time, the DStreams used for the operation to be performed are enclosed in a rectangle.

In every batch interval, a new DStream is generated. Here, the window length is three and the operation to be performed in a window is counting the number of event messages in that window. The sliding interval is kept the same as the batch interval so that the counting operation is done as and when a new DStream is generated, so that the count is correct all the time.

At time **t2**, the counting operation is done on the DStreams generated at times **t0**, **t1**, and **t2**. At time **t3**, the counting operation is done again since the sliding window is kept the same as the batch interval, and this time counting the events is done on the DStreams generated at time **t1**, **t2**, and **t3**. At time **t4**, the counting operation is done again, counting the events done on the DStreams generated at time **t2**, **t3**, and **t4**. The operations continue in that fashion till the application is terminated.

Counting the number of log event messages processed in Scala

In the preceding section, the processing of the log event messages is discussed. In the same application code after the printing of the log event messages containing the word ERROR, include the following lines of code in the Scala application:

```
errorLines.print()
errorLines.countByWindow(Seconds(30), Seconds(10)).print()
```

The first parameter is the window length and the second one is the sliding window interval. This single magic line will print a count of log event messages processed once the following lines are typed in the Netcat console:

```
[Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:46:23 2015] [WARN] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /apache/web/test
```

The same Spark Streaming data processing application in Scala, with the additional lines of code, produces the following output:

```
-------------------------------------------

Time: 1469284630000 ms
```

```
     ------------------------------------------
     [Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory
index
     forbidden by rule: /home/raj/
     ------------------------------------------
     Time: 1469284630000 ms
     ------------------------------------------
     1
     ------------------------------------------
     Time: 1469284640000 ms
     ------------------------------------------
     [Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory
index
     forbidden by rule: /apache/web/test
     ------------------------------------------
     Time: 1469284640000 ms
     ------------------------------------------
     2
     ------------------------------------------
     Time: 1469284650000 ms
     ------------------------------------------
     2
     ------------------------------------------
     Time: 1469284660000 ms
     ------------------------------------------
     1
     ------------------------------------------
     Time: 1469284670000 ms
     ------------------------------------------
```

If the output is studied properly, it can be noticed that, in the first batch interval, one log event message is processed. Obviously the count displayed is 1 for that batch interval. In the next batch interval, one more log event message is processed. The count displayed for that batch interval is 2. In the next batch interval, no log event message is processed. But the count for that window is still 2. For one more window, the count is displayed as 2. Then it reduces to 1, and then 0.

The most important point to be noted here is that, in the application codes for both Scala and Python, immediately after StreamingContext creation, the following line of code needs to be inserted to specify the checkpoint directory:

```
ssc.checkpoint("/tmp")
```

Counting the number of log event messages processed in Python

In the Python application code, after the printing of the log event messages containing the word ERROR, include the following lines of code in the Scala application:

```
errorLines.pprint()
errorLines.countByWindow(30,10).pprint()
```

The first parameter is the window length and the second one is the sliding window interval. This single magic line will print a count of log event messages processed once the following lines are typed in the Netcat console:

```
[Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:46:23 2015] [WARN] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /apache/web/test
```

The same Spark Streaming data processing application in Python, with the additional lines of code, produces the following output:

```
-------------------------------------------
Time: 2016-07-23 15:29:40
-------------------------------------------
[Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
-------------------------------------------
Time: 2016-07-23 15:29:40
-------------------------------------------
1
-------------------------------------------
Time: 2016-07-23 15:29:50
-------------------------------------------
[Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /apache/web/test
-------------------------------------------
Time: 2016-07-23 15:29:50
-------------------------------------------
2
-------------------------------------------
Time: 2016-07-23 15:30:00
-------------------------------------------
-------------------------------------------
Time: 2016-07-23 15:30:00
```

```
------------------------------------------
2
------------------------------------------
Time: 2016-07-23 15:30:10
------------------------------------------
------------------------------------------
Time: 2016-07-23 15:30:10
------------------------------------------
1
------------------------------------------
Time: 2016-07-23 15:30:20
------------------------------------------
------------------------------------------
Time: 2016-07-23 15:30:20
------------------------------------------
```

The output pattern of the Python application is also very similar to the Scala application.

More processing options

Apart from the count operation in a window, there are more operations that can be done on DStreams in conjunction with windowing. The following table captures the important transformations. All these transformations are acting on the selected window and return a DStream.

Transformation	Description
window(windowLength, slideInterval)	Returns DStreams computed in the window
countByWindow(windowLength, slideInterval)	Returns the count of elements
reduceByWindow(func, windowLength, slideInterval)	Returns one element by applying the aggregation function
reduceByKeyAndWindow(func, windowLength, slideInterval, [numTasks])	Returns one key/value pair per key after applying the aggregation function over multiple values per key

countByValueAndWindow(windowLength, slideInterval, [numTasks])	Returns one key/count pair per key after applying the count of multiple values per key

One of the most important steps of stream processing is the persisting of the stream data into secondary storage. Since the velocity of the data in Spark Streaming data processing applications is going to be very high, any kind of persistence mechanism that introduces additional latency into the whole process is not an advisable solution.

In batch processing scenarios, it is fine to write to the HDFS and other file system based storage. But when it comes to the storage of stream output, depending on the use case, an ideal stream data storage mechanism should be chosen.

NoSQL data stores such as Cassandra support fast writes of temporal data. It is also ideal to read the data that is stored for any further analysis purposes. Spark Streaming library supports many output methods on DStreams. They include options to save the stream data as text file, object file, Hadoop files, and so on. As well as this, there are many third-party drivers available to save the data into various data stores.

Kafka stream processing

The log event processor example covered in this chapter was listening to a TCP socket for the stream of messages to be processed by the Spark Streaming data processing application. But in real-world use cases, this is not going to be the case.

Message queueing systems with publish-subscribe capability are generally used for processing messages. The traditional message queueing systems failed to perform because of the huge volume of messages to be processed per second for the needs of large-scale data processing applications.

Kafka is a publish-subscribe messaging system used by many IoT applications to process a huge number of messages. The following capabilities of Kafka made it one of the most widely used messaging systems:

- Extremely fast: Kafka can process huge amounts of data by handling reading and writing in short intervals of time from many application clients
- Highly scalable: Kafka is designed to scale up and scale out to form a cluster using commodity hardware

- Persists a huge number of messages: Messages reaching Kafka topics are persisted into the secondary storage, while at the same time it is handling huge number of messages flowing through

> A detailed treatment of Kafka is outside the scope of this book. It is assumed that the reader is familiar with and has working knowledge of Kafka. From a Spark Streaming data processing application perspective, it doesn't really make a difference whether it is a TCP socket or Kafka that is being used as a message source. But working on a teaser use case with Kafka as the message producer will give a good appreciation of the toolsets enterprises are using heavily. *Learning Apache Kafka – Second Edition* by *Nishant Garg* (`https://www.packtpub.com/big-data-and-bus iness-intelligence/learning-apache-kafka-second-edition`) is a good reference book to learn more about Kafka.

The following are some of the important elements of Kafka, and are terms to be understood before proceeding further:

- Producer: The real source of the messages, such as weather sensors or mobile phone network
- Broker: The Kafka cluster, which receives and persists the messages published to its topics by various producers
- Consumer: The data processing applications subscribed to the Kafka topics that consume the messages published to the topics

The same log event processing application use case discussed in the preceding section is used again here to elucidate the usage of Kafka with Spark Streaming. Instead of collecting the log event messages from the TCP socket, here the Spark Streaming data processing application will act as a consumer of a Kafka topic and the messages published to the topic will be consumed.

The Spark Streaming data processing application uses the version 0.8.2.2 of Kafka as the message broker, and the assumption is that the reader has already installed Kafka, at least in a standalone mode. The following activities are to be performed to make sure that Kafka is ready to process the messages produced by the producers and that the Spark Streaming data processing application can consume those messages:

1. Start the Zookeeper that comes with Kafka installation.
2. Start the Kafka server.
3. Create a topic for the producers to send the messages to.

4. Pick up one Kafka producer and start publishing log event messages to the newly created topic.

5. Use the Spark Streaming data processing application to process the log events published to the newly created topic.

Starting Zookeeper and Kafka

The following scripts are run from separate terminal windows in order to start Zookeeper and the Kafka broker, and to create the required Kafka topics:

```
$ cd $KAFKA_HOME
$ $KAFKA_HOME/bin/zookeeper-server-start.sh
$KAFKA_HOME/config/zookeeper.properties
[2016-07-24 09:01:30,196] INFO binding to port 0.0.0.0/0.0.0.0:2181
(org.apache.zookeeper.server.NIOServerCnxnFactory)
$ $KAFKA_HOME/bin/kafka-server-start.sh
$KAFKA_HOME/config/server.properties

[2016-07-24 09:05:06,381] INFO 0 successfully elected as leader
(kafka.server.ZookeeperLeaderElector)
[2016-07-24 09:05:06,455] INFO [Kafka Server 0], started
(kafka.server.KafkaServer)
$ $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 --
replication-factor 1 --partitions 1 --topic sfb
Created topic "sfb".
$ $KAFKA_HOME/bin/kafka-console-producer.sh --broker-list localhost:9092 --
topic sfb
```

> **TIP**
> Make sure that the environment variable $KAFKA_HOME is pointing to the directory where Kafka is installed. Also, it is very important to start Zookeeper, Kafka server, Kafka producer, and Spark Streaming log event data processing application in separate terminal windows.

The Kafka message producer can be any application capable of publishing messages to the Kafka topics. Here, the kafka-console-producer that comes with Kafka is used as the producer of choice. Once the producer starts running, whatever is typed into its console window will be treated as a message that is published to the chosen Kafka topic. The Kafka topic is given as a command line argument when starting the kafka-console-producer.

The submission of the Spark Streaming data processing application that consumes log event messages produced by the Kafka producer is slightly different from the application covered in the preceding section. Here, many Kafka jar files are required for the data processing. Since they are not part of the Spark infrastructure, they have to be submitted to the Spark cluster. The following jar files are required for the successful running of this application:

- `$KAFKA_HOME/libs/kafka-clients-0.8.2.2.jar`
- `$KAFKA_HOME/libs/kafka_2.11-0.8.2.2.jar`
- `$KAFKA_HOME/libs/metrics-core-2.2.0.jar`
- `$KAFKA_HOME/libs/zkclient-0.3.jar`
- `Code/Scala/lib/spark-streaming-kafka-0-8_2.11-2.0.0-preview.jar`
- `Code/Python/lib/spark-streaming-kafka-0-8_2.11-2.0.0-preview.jar`

In the preceding list of jar files, the maven repository co-ordinate for `spark-streaming-kafka-0-8_2.11-2.0.0-preview.jar` is `"org.apache.spark" %% "spark-streaming-kafka-0-8" % "2.0.0-preview"`. This particular jar file has to be downloaded and placed in the lib folder of the directory structure given in Figure 4. It is being used in the `submit.sh` and the `submitPy.sh` scripts, which submit the application to the Spark cluster. The download URL for this jar file is given in the reference section of this chapter.

In the `submit.sh` and `submitPy.sh` files, the last few lines contain a conditional statement looking for the second parameter value of 1 to identify this application and ship the required jar files to the Spark cluster.

> **TIP**
>
> Instead of shipping these individual jar files separately to the Spark cluster when the job is submitted, an assembly jar can be used by creating it using sbt.

Implementing the application in Scala

The following code snippet is the Scala code for the log event processing application that processes the messages produced by the Kafka producer. The use case of this application is the same as the one discussed in the preceding section concerning windowing operations:

```
/**
The following program can be compiled and run using SBT
Wrapper scripts have been provided with this
The following script can be run to compile the code
```

```
 ./compile.sh

The following script can be used to run this application in Spark. The
second command line argument of value 1 is very important. This is to flag
the shipping of the kafka jar files to the Spark cluster
./submit.sh com.packtpub.sfb.KafkaStreamingApps 1
**/
package com.packtpub.sfb

import java.util.HashMap
import org.apache.spark.streaming._
import org.apache.spark.sql.{Row, SparkSession}
import org.apache.spark.streaming.kafka._
import org.apache.kafka.clients.producer.{ProducerConfig, KafkaProducer,
ProducerRecord}

object KafkaStreamingApps {
  def main(args: Array[String]) {
   // Log level settings
   LogSettings.setLogLevels()
   // Variables used for creating the Kafka stream
   //The quorum of Zookeeper hosts
    val zooKeeperQuorum = "localhost"
   // Message group name
   val messageGroup = "sfb-consumer-group"
   //Kafka topics list separated by coma if there are multiple topics to be
listened on
   val topics = "sfb"
   //Number of threads per topic
   val numThreads = 1
   // Create the Spark Session and the spark context
   val spark = SparkSession
         .builder
         .appName(getClass.getSimpleName)
         .getOrCreate()
   // Get the Spark context from the Spark session for creating the
streaming context
   val sc = spark.sparkContext
   // Create the streaming context
   val ssc = new StreamingContext(sc, Seconds(10))
    // Set the check point directory for saving the data to recover when
there is a crash
   ssc.checkpoint("/tmp")
   // Create the map of topic names
    val topicMap = topics.split(",").map((_, numThreads.toInt)).toMap
   // Create the Kafka stream
    val appLogLines = KafkaUtils.createStream(ssc, zooKeeperQuorum,
messageGroup, topicMap).map(_._2)
```

```
   // Count each log messge line containing the word ERROR
    val errorLines = appLogLines.filter(line => line.contains("ERROR"))
   // Print the line containing the error
   errorLines.print()
   // Count the number of messages by the windows and print them
   errorLines.countByWindow(Seconds(30), Seconds(10)).print()
   // Start the streaming
    ssc.start()
   // Wait till the application is terminated
    ssc.awaitTermination()
  }
}
```

Compared to the Scala code in the preceding section, the major difference is in the way the stream is created.

Implementing the application in Python

The following code snippet is the Python code for the log event processing application that processes the message produced by the Kafka producer. The use case of this application is also the same as the one discussed in the preceding section concerning windowing operations:

```
# The following script can be used to run this application in Spark
# ./submitPy.sh KafkaStreamingApps.py 1

from __future__ import print_function
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
from pyspark.streaming.kafka import KafkaUtils

if __name__ == "__main__":
    # Create the Spark context
    sc = SparkContext(appName="PythonStreamingApp")
    # Necessary log4j logging level settings are done
    log4j = sc._jvm.org.apache.log4j
    log4j.LogManager.getRootLogger().setLevel(log4j.Level.WARN)
    # Create the Spark Streaming Context with 10 seconds batch interval
    ssc = StreamingContext(sc, 10)
    # Set the check point directory for saving the data to recover when
there is a crash
    ssc.checkpoint("\tmp")
    # The quorum of Zookeeper hosts
    zooKeeperQuorum="localhost"
```

```
    # Message group name
    messageGroup="sfb-consumer-group"
    # Kafka topics list separated by coma if there are multiple topics to
be listened on
    topics = "sfb"
    # Number of threads per topic
    numThreads = 1
    # Create a Kafka DStream
    kafkaStream = KafkaUtils.createStream(ssc, zooKeeperQuorum,
messageGroup, {topics: numThreads})
    # Create the Kafka stream
    appLogLines = kafkaStream.map(lambda x: x[1])
    # Count each log messge line containing the word ERROR
    errorLines = appLogLines.filter(lambda appLogLine: "ERROR" in
appLogLine)
    # Print the first ten elements of each RDD generated in this DStream to
the console
    errorLines.pprint()
    errorLines.countByWindow(30,10).pprint()
    # Start the streaming
    ssc.start()
    # Wait till the application is terminated
    ssc.awaitTermination()
```

The following commands are run on the terminal window to run the Scala application:

```
$ cd Scala
$ ./submit.sh com.packtpub.sfb.KafkaStreamingApps 1
```

The following commands are run on the terminal window to run the Python application:

```
$ cd Python
$
./submitPy.sh KafkaStreamingApps.py 1
```

When both of the preceding programs are running, whatever log event messages are typed into the console window of the Kafka console producer, and invoked using the following command and inputs, will be processed by the application. The outputs of this program will be very similar to the ones that are given in the preceding section:

```
$ $KAFKA_HOME/bin/kafka-console-producer.sh --broker-list localhost:9092 --
topic sfb
[Fri Dec 20 01:46:23 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:46:23 2015] [WARN] [client 1.2.3.4.5.6] Directory index
forbidden by rule: /home/raj/
[Fri Dec 20 01:54:34 2015] [ERROR] [client 1.2.3.4.5.6] Directory index
```

```
forbidden by rule: /apache/web/test
```

Spark provides two approaches to process Kafka streams. The first one is the receiver-based approach that was discussed previously and the second one is the direct approach.

This direct approach to processing Kafka messages is a simplified method in which Spark Streaming is using all the possible capabilities of Kafka just like any of the Kafka topic consumers, and polls for the messages in the specific topic, and the partition by the offset number of the messages. Depending on the batch interval of the Spark Streaming data processing application, it picks up a certain number of offsets from the Kafka cluster, and this range of offsets is processed as a batch. This is highly efficient and ideal for processing messages with a requirement to have exactly-once processing. This method also reduces the Spark Streaming library's need to do additional work to implement the exactly-once semantics of the message processing and delegates that responsibility to Kafka. The programming constructs of this approach are slightly different in the APIs used for the data processing. Consult the appropriate reference material for the details.

The preceding sections introduced the concept of a Spark Streaming library and discussed some of the real-world use cases. There is a big difference between Spark data processing applications developed to process static batch data and those developed to process dynamic stream data in a deployment perspective. The availability of data processing applications to process a stream of data must be constant. In other words, such applications should not have components that are single points of failure. The following section is going to discuss this topic.

Spark Streaming jobs in production

When a Spark Streaming application is processing the incoming data, it is very important to have uninterrupted data processing capability so that all the data that is getting ingested is processed. In business-critical streaming applications, most of the time missing even one piece of data can have a huge business impact. To deal with such situations, it is important to avoid single points of failure in the application infrastructure.

From a Spark Streaming application perspective, it is good to understand how the underlying components in the ecosystem are laid out so that the appropriate measures can be taken to avoid single points of failure.

A Spark Streaming application deployed in a cluster such as Hadoop YARN, Mesos or Spark Standalone mode has two main components very similar to any other type of Spark application:

- **Spark driver**: This contains the application code written by the user
- **Executors**: The executors that execute the jobs submitted by the Spark driver

But the executors have an additional component called a receiver that receives the data getting ingested as a stream and saves it as blocks of data in memory. When one receiver is receiving the data and forming the data blocks, they are replicated to another executor for fault-tolerance. In other words, in-memory replication of the data blocks is done onto a different executor. At the end of every batch interval, these data blocks are combined to form a DStream and sent out for further processing downstream.

Figure 8 depicts the components working together in a Spark Streaming application infrastructure deployed in a cluster:
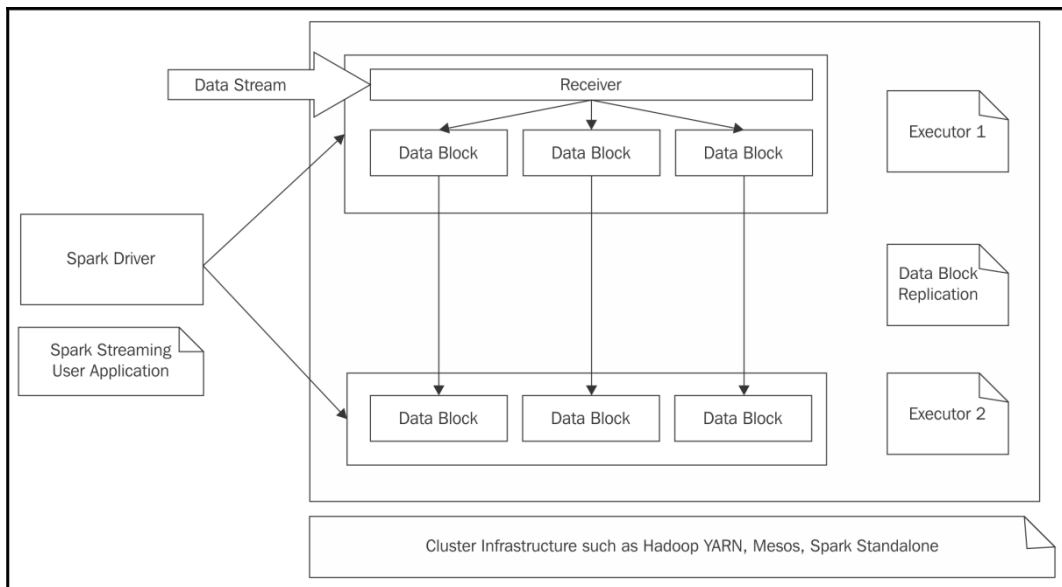
Figure 8

In *Figure 8*, there are two executors. The receiver component is deliberately not displayed in the second executor to show that it is not using the receiver and instead just collects the replicated data blocks from the other executor. But when needed, such as on the failure of the first executor, the receiver in the second executor can start functioning.

Implementing fault-tolerance in Spark Streaming data processing applications

Spark Streaming data processing application infrastructure has many moving parts. Failures can happen to any one of them, resulting in the interruption of the data processing. Typically failures can happen to the Spark driver or the executors.

> This section is not intended to give detailed treatment to running the Spark Streaming applications in production with fault-tolerance. The intention is to make the reader appreciate the precautions to be taken when deploying Spark Streaming data processing applications in production.

When an executor fails, since the replication of data is happening on a regular basis, the task of receiving the data stream will be taken over by the executor on which the data was getting replicated. There is a situation in which when an executor fails, all the data that is unprocessed will be lost. To circumvent this problem, there is a way to persist the data blocks into HDFS or Amazon S3 in the form of write-ahead logs.

> There is no need to have both the in-memory replication of the data blocks and write-ahead logs together in one infrastructure. Keep only one of them, depending on the need.

When the Spark driver fails, the driven program is stopped, all the executors lose connection, and they stop functioning. This is the most dangerous situation. To deal with this situation, some configuration and code changes are necessary.

The Spark driver has to be configured to have an automatic driver restart, which is supported by the cluster managers. This includes a change in the Spark job submission method to have the cluster mode in whichever may be the cluster manager. When a restart of the driver happens, to start from the place when it crashed, a checkpointing mechanism has to be implemented in the driver program. This has already been done in the code samples that are used. The following lines of code do that job:

```
ssc = StreamingContext(sc, 10)
ssc.checkpoint("\tmp")
```

> In a sample application, it is fine to use a local system directory as the checkpoint directory. But in a production environment, it is better to keep this checkpoint directory as an HDFS location in the case of Hadoop or an S3 location in the case of an Amazon cloud.

From an application coding perspective, the way the `StreamingContext` is created is slightly different. Instead of creating a new `StreamingContext` every time, the factory method `getOrCreate` of the `StreamingContext` is to be used with a function, as shown in the following code segment. If that is done, when the driver is restarted, the factory method will check the checkpoint directory to see whether an earlier `StreamingContext` was in use, and, if found in the checkpoint data, it is created. Otherwise, a new `StreamingContext` is created.

The following code snippet gives the definition of a function that can be used with the `getOrCreate` factory method of the `StreamingContext`. As mentioned earlier, a detailed treatment of these aspects is beyond the scope of this book:

```
  /**
   * The following function has to be used when the code is being
restructured to have checkpointing and driver recovery
   * The way it should be used is to use the StreamingContext.getOrCreate
with this function and do a start of that
   */
  def sscCreateFn(): StreamingContext = {
   // Variables used for creating the Kafka stream
   // The quorum of Zookeeper hosts
    val zooKeeperQuorum = "localhost"
   // Message group name
   val messageGroup = "sfb-consumer-group"
   //Kafka topics list separated by coma if there are multiple topics to be
listened on
   val topics = "sfb"
   //Number of threads per topic
   val numThreads = 1
   // Create the Spark Session and the spark context
   val spark = SparkSession
        .builder
        .appName(getClass.getSimpleName)
        .getOrCreate()
   // Get the Spark context from the Spark session for creating the
streaming context
   val sc = spark.sparkContext
   // Create the streaming context
   val ssc = new StreamingContext(sc, Seconds(10))
   // Create the map of topic names
    val topicMap = topics.split(",").map((_, numThreads.toInt)).toMap
   // Create the Kafka stream
    val appLogLines = KafkaUtils.createStream(ssc, zooKeeperQuorum,
messageGroup, topicMap).map(_._2)
   // Count each log messge line containing the word ERROR
    val errorLines = appLogLines.filter(line => line.contains("ERROR"))
```

```
    // Print the line containing the error
    errorLines.print()
    // Count the number of messages by the windows and print them
    errorLines.countByWindow(Seconds(30), Seconds(10)).print()
    // Set the check point directory for saving the data to recover when
  there is a crash
    ssc.checkpoint("/tmp")
    // Return the streaming context
    ssc
  }
```

At a data source level, it is a good idea to build parallelism for faster data processing and, depending on the source of data, this can be accomplished in different ways. Kafka inherently supports partition at the topic level, and that kind of scaling out mechanism supports a good amount of parallelism. As a consumer of Kafka topics, the Spark Streaming data processing application can have multiple receivers by creating multiple streams, and the data generated by those streams can be combined by the union operation on the Kafka streams.

The production deployment of Spark Streaming data processing applications is to be done purely based on the type of application that is being used. Some of the guidelines given previously are just introductory and conceptual in nature. There is no silver bullet approach to solving production deployment problems, and they have to evolve along with the application development.

Structured streaming

In the data streaming use cases that have been covered so far, there are many developer tasks in terms of building of the structure data and implementing fault tolerance for the application. The data that has been dealt with so far in data streaming applications is unstructured data. Just like the batch data processing use cases, even in streaming use cases, if there is the capability to process structured data, that is a great advantage, and lots of pre-processing can be avoided. Data stream processing applications are continuously running applications and they are bound to develop failures or interruptions. In such situations, it is imperative to build fault tolerance in the data streaming applications.

In any data streaming application, the data is getting ingested continuously, and if there is a need to interrogate the data received at any given point in time, the application developers have to persist the data processed into a data store that supports querying. In Spark 2.0, the structured streaming concept is built around these aspects, and the whole idea behind building this brand new feature from the ground up is to relieve application developers of these pain areas. There is a feature with the reference number SPARK-8360 being built at the time of writing this chapter, and its progress can be monitored by visiting the corresponding page.

The structured streaming concept can be explained using a real-world use case, such as the banking transaction use case we looked at before. Assume that the comma-separated transaction records containing the account number and transaction amount are coming in a stream. In the structured stream processing method, all these data items get ingested into an unbounded table or DataFrame that supports querying using Spark SQL. In other words, since the data is accumulated in a DataFrame, whatever data processing is possible using a DataFrame will be possible with the stream data as well. This reduces the burden on application developers and they can focus on the business logic of the application rather than the infrastructure related-aspects.

References

For more information, visit the following:

- https://spark.apache.org/docs/latest/streaming-programming-guide.html
- http://kafka.apache.org/
- http://spark.apache.org/docs/latest/streaming-kafka-integration.html
- https://www.packtpub.com/big-data-and-business-intelligence/learning-apache-kafka-second-edition
- http://search.maven.org/remotecontent?filepath=org/apache/spark/spark-streaming-kafka--8_2.11/2..-preview/spark-streaming-kafka--8_2.11-2..-preview.jar
- https://issues.apache.org/jira/browse/SPARK-836

Summary

Spark provides a very powerful library on top of the Spark core to process the stream of data getting ingested at a high velocity. This chapter introduced the basics of the Spark Streaming library, and a simple log event message processing system has been developed with two types of data source: one uses a TCP data server and the other uses Kafka. At the end of the chapter, a brief look at the production deployment of Spark Streaming data processing applications is provided and the possible ways of implementing fault-tolerance in Spark Streaming data processing applications as discussed.

Spark 2.0 brings the capability to process and query structured data in streaming applications, and the concept has been introduced, which relieves application developers from pre-processing the unstructured data, building fault-tolerance and querying the data that is being ingested on a near-real-time basis.

Applied mathematicians and statisticians have come up with ways and means to answer questions related to a new piece of data based on the *learning* that has already been done on an existing bank of data. Typically these questions include, but are not limited to: does this piece of data fit a given model, can this piece of data be classified in a certain way, and does this piece of data belong to any group or cluster?

There are lots of algorithms available to *train* a data model and ask questions to this *model* about the new piece of data. This rapidly evolving branch of data science has huge applicability in data processing, and is popularly known as machine learning. The next chapter is going to discuss the machine learning library of Spark.

7
Spark Machine Learning

Calculations based on formulas or algorithms have been used commonly since ancient times to find the output for a given input. But without knowing the formulas or algorithms, computer scientists and mathematicians devised methods to generate formulas or algorithms based on an existing input/output dataset and predicted the output of a new input data based on the generated formulas or algorithms. Generally, this process of learning from a dataset and doing predictions based on the learning is known as machine learning. Machine learning originates from the study of artificial intelligence in computer science.

Practical machine learning has numerous applications that are being consumed by laymen on a daily basis. YouTube users now get suggestions for the next items to be played in the playlist based on the video they are currently viewing. Popular movie rating sites give ratings and recommendations based on the user preferences of movie genres. Social media web sites such as Facebook suggest a list of names of the users' friends for easy tagging of pictures. What Facebook is doing here is classifying pictures by the names that are already available in the existing albums and checking whether the newly added picture has any similarities with the existing ones. If it finds a similarity, it suggests the name. The applications of this kind of picture identification are manifold. The way all these applications work is based on the huge amount of input/output datasets that have already been collected and the learning that has been done based on those datasets. When a new input dataset arrives, a prediction is made by making use of the learning that the computer or machine has already done.

We will cover the following topics in this chapter:

- Using Spark for machine learning
- Model persistence
- Spam filtering

- Feature algorithms
- Finding synonyms

Understanding machine learning

In traditional computing, input data is fed to a program to generate output. But in machine learning, input data and output data are fed to a machine learning algorithm to generate a function or program that can be used to predict the output of an input according to the learning done on the input/output dataset fed to the machine learning algorithm.

The data available in the wild may be classified into groups, it may form clusters, or it may fit into certain relationships. These are different kinds of machine learning problem. For example, if there is a databank of pre-owned car sale prices with its associated attributes or features, it is possible to predict the price of a car just by knowing the associated attributes or features. Regression algorithms are used to solve these kinds of problem. If there is a databank of spam and non-spam e-mails, then when a new e-mail comes, it is possible to predict whether the new e-mail is spam or non-spam. Classification algorithms are used to solve these kinds of problem.

These are just a few machine learning algorithm types. But in general, when using a bank of data, if it is necessary to apply a machine learning algorithm and use that model to make predictions, then the data should be divided into features and outputs. For example, in the case of the car price prediction problem, price is the output, and here are some of the possible features of the data:

- Car make
- Car model
- Year of manufacture
- Mileage
- Fuel type
- Gearbox type

So whichever machine learning algorithm is being used, there will be a set of features and one or more outputs.

> Many books and publications use the term *label* for output. In other words, *features* are the input and *label* is the output.

Figure 1 depicts the way a machine learning algorithm works on the underlying data to enable predictions.
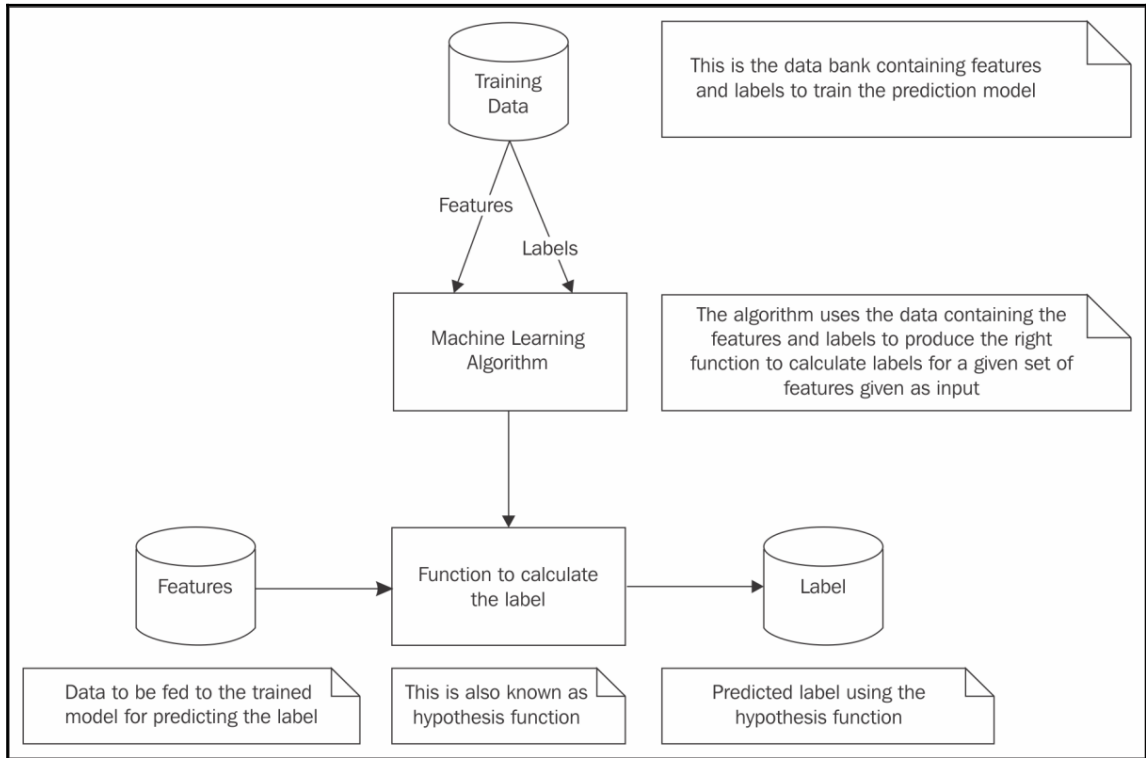
Figure 1

Data comes in various shapes and forms. Depending on the machine learning algorithm used, the training data has to be pre-processed to have the features and labels in the right format to be fed to the machine learning algorithm. That in turn generates the appropriate hypothesis function, which takes the features as the input and produces the predicted label.

> The dictionary definition of the word hypothesis is a supposition or proposed explanation made on the basis of limited evidence as a starting point for further investigation. Here, the function or program that is generated by the machine learning algorithm is based on the limited evidence that is the training data fed to the machine learning algorithm, and hence it is widely known as hypothesis function.
>
> In other words, this hypothesis function is not a definitive function that produces consistent results all the time with all types of input data. It is

> **TIP**
>
> rather a function based on the training data. When a new piece of data is added to the training dataset, re-learning is required, and at that time even the hypothesis function generated will change accordingly.

In reality, the flow given in *Figure 1* is not as simple as it seems. Once the model is trained, a lot of testing has to be done on the model to test predictions with known labels. The chain of train and test processes is an iterative process, and in each iteration the parameters of the algorithm are tweaked to make the prediction quality better. Once an acceptable test result is produced by the model, the model can be moved to production for doing the live prediction needs. Spark comes with a machine learning librarythat is rich with capabilities to make practical machine learning a reality.

Why Spark for machine learning?

The previous chapters covered various data processing functionalities of Spark in detail. Spark's machine learning library uses many Spark core functionalities as well as Spark libraries such as Spark SQL. The Spark machine learning library makes machine learning application development easy by combining data processing and machine learning algorithm implementations in a unified framework with the ability to do data processing on a cluster of nodes, combined with ability to read and write data to a variety of data formats.

Spark comes with two flavors of the machine learning library. They are `spark.mllib` and `spark.ml`. The first one is developed on top of Spark's RDD abstraction, and the second one is developed on top of Spark's DataFrame abstraction. It is recommended to use the spark.ml library for any future machine learning application developments.

This chapter is going to focus only on the spark.ml machine learning library. The following list explains terminology and concepts that are used again and again in this chapter:

- **Estimator**: This is an algorithm that works on top of a Spark DataFrame containing features and labels. It trains on the data provided in the Spark DataFrame and creates a model. This model is used to do future predictions.
- **Transformer**: This converts a Spark DataFrame containing features and transforms it to another Spark DataFrame containing predictions. The model created by an Estimator is a Transformer.
- **Parameter**: This is to be used by the Estimators and Transformers. Often, this is specific to the machine learning algorithm. Spark machine learning library comes with a uniform API for specifying the right parameters to the algorithms.
- **Pipeline**: This is a chain of Estimators and Transformers working together forming a machine learning workflow.

All these new terms are slightly difficult to understand in a theoretical perspective but if an example is given, the concepts will become much clearer.

Wine quality prediction

The University of California Irvine Machine Learning Repository (`http://archive.ics.uci.edu/ml/index.html`) provides a lot of datasets as a service to those who are interested in learning about machine learning. The Wine Quality Dataset (`http://archive.ics.uci.edu/ml/datasets/Wine+Quality`
) is being used here to demonstrate some machine learning applications. It contains two datasets with various features of white and red wines from Portugal.

> The Wine Quality Dataset download link lets you download the datasets for red wine and white wine as two separate CSV files. Once those files are downloaded, edit the two datasets to remove the first header line containing the column names. This is to let the programs parse the numerical data without errors. Detailed error handling and excluding the header record are avoided on purpose to focus on the machine learning functionality.

The dataset containing various features of red wine is used in this wine quality prediction use case. The following are the features of the dataset:

- Fixed acidity
- Volatile acidity
- Citric acid
- Residual sugar
- Chlorides
- Free sulfur dioxide
- Total sulfur dioxide
- Density
- pH
- Sulphates
- Alcohol

Based on these features, the quality (score between 0 and 10) is determined. Here, quality is the label of this dataset. Using this dataset, a model is going to be trained and then, using the trained model, testing is done and predictions are made. This is a regression problem. The Linear Regression algorithm is used to train the model. The Linear Regression algorithm generates a linear hypothesis function. In mathematical terms, a linear function is a polynomial of degree one or less. In this machine learning application use case, it deals with modeling the relationship between a dependent variable (wine quality) and a set of independent variables (the features of the wine).

At the Scala REPL prompt, try the following statements:

```scala
scala> import org.apache.spark.ml.regression.LinearRegression
    import org.apache.spark.ml.regression.LinearRegression
scala> import org.apache.spark.ml.param.ParamMap
    import org.apache.spark.ml.param.ParamMap
scala> import org.apache.spark.ml.linalg.{Vector, Vectors}
    import org.apache.spark.ml.linalg.{Vector, Vectors}
scala> import org.apache.spark.sql.Row
    import org.apache.spark.sql.Row
scala> // TODO – Change this directory to the right location where the
data
is stored
scala> val dataDir = "/Users/RajT/Downloads/wine-quality/"
    dataDir: String = /Users/RajT/Downloads/wine-quality/
scala> // Define the case class that holds the wine data
scala> case class Wine(FixedAcidity: Double, VolatileAcidity: Double,
CitricAcid: Double, ResidualSugar: Double, Chlorides: Double,
FreeSulfurDioxide: Double, TotalSulfurDioxide: Double, Density: Double, PH:
Double, Sulphates: Double, Alcohol: Double, Quality: Double)
    defined class Wine
scala> // Create the the RDD by reading the wine data from the disk
scala> //TODO – The wine data has to be downloaded to the appropriate
working directory in the system where this is being run and the following
line of code should use that path
scala> val wineDataRDD = sc.textFile(dataDir + "winequality-
red.csv").map(_.split(";")).map(w => Wine(w(0).toDouble, w(1).toDouble,
w(2).toDouble, w(3).toDouble, w(4).toDouble, w(5).toDouble, w(6).toDouble,
w(7).toDouble, w(8).toDouble, w(9).toDouble, w(10).toDouble,
w(11).toDouble))
    wineDataRDD: org.apache.spark.rdd.RDD[Wine] = MapPartitionsRDD[3] at
map at <console>:32
scala> // Create the data frame containing the training data having two
columns. 1) The actual output or label of the data 2) The vector containing
the features
scala> //Vector is a data type with 0 based indices and double-typed
values. In that there are two types namely dense and sparse.
scala> //A dense vector is backed by a double array representing its
entry values
```

```
    scala> //A sparse vector is backed by two parallel arrays: indices and
values
    scala> val trainingDF = wineDataRDD.map(w => (w.Quality,
Vectors.dense(w.FixedAcidity, w.VolatileAcidity, w.CitricAcid,
w.ResidualSugar, w.Chlorides, w.FreeSulfurDioxide, w.TotalSulfurDioxide,
w.Density, w.PH, w.Sulphates, w.Alcohol))).toDF("label", "features")
      trainingDF: org.apache.spark.sql.DataFrame = [label: double,
features: vector]
    scala> trainingDF.show()
      +-----+--------------------+
      |label|            features|
      +-----+--------------------+
      |  5.0|[7.4,0.7,0.0,1.9,...|
      |  5.0|[7.8,0.88,0.0,2.6...|
      |  5.0|[7.8,0.76,0.04,2....|
      |  6.0|[11.2,0.28,0.56,1...|
      |  5.0|[7.4,0.7,0.0,1.9,...|
      |  5.0|[7.4,0.66,0.0,1.8...|
      |  5.0|[7.9,0.6,0.06,1.6...|
      |  7.0|[7.3,0.65,0.0,1.2...|
      |  7.0|[7.8,0.58,0.02,2....|
      |  5.0|[7.5,0.5,0.36,6.1...|
      |  5.0|[6.7,0.58,0.08,1....|
      |  5.0|[7.5,0.5,0.36,6.1...|
      |  5.0|[5.6,0.615,0.0,1....|
      |  5.0|[7.8,0.61,0.29,1....|
      |  5.0|[8.9,0.62,0.18,3....|
      |  5.0|[8.9,0.62,0.19,3....|
      |  7.0|[8.5,0.28,0.56,1....|
      |  5.0|[8.1,0.56,0.28,1....|
      |  4.0|[7.4,0.59,0.08,4....|
      |  6.0|[7.9,0.32,0.51,1....|
      +-----+--------------------+
      only showing top 20 rows
    scala> // Create the object of the algorithm which is the Linear
Regression
    scala> val lr = new LinearRegression()
      lr: org.apache.spark.ml.regression.LinearRegression =
linReg_f810f0c1617b
    scala> // Linear regression parameter to make lr.fit() use at most 10
iterations
    scala> lr.setMaxIter(10)
      res1: lr.type = linReg_f810f0c1617b
    scala> // Create a trained model by fitting the parameters using the
training data
    scala> val model = lr.fit(trainingDF)
      model: org.apache.spark.ml.regression.LinearRegressionModel =
linReg_f810f0c1617b
```

```
    scala> // Once the model is prepared, to test the model, prepare the
test data containing the labels and feature vectors
    scala> val testDF = spark.createDataFrame(Seq((5.0, Vectors.dense(7.4,
0.7, 0.0, 1.9, 0.076, 25.0, 67.0, 0.9968, 3.2, 0.68,9.8)),(5.0,
Vectors.dense(7.8, 0.88, 0.0, 2.6, 0.098, 11.0, 34.0, 0.9978, 3.51, 0.56,
9.4)),(7.0, Vectors.dense(7.3, 0.65, 0.0, 1.2, 0.065, 15.0, 18.0, 0.9968,
3.36, 0.57, 9.5)))).toDF("label", "features")
    testDF: org.apache.spark.sql.DataFrame = [label: double, features:
vector]
    scala> testDF.show()
      +-----+--------------------+
      |label|            features|
      +-----+--------------------+
      |  5.0|[7.4,0.7,0.0,1.9,...|
      |  5.0|[7.8,0.88,0.0,2.6...|
      |  7.0|[7.3,0.65,0.0,1.2...|
      +-----+--------------------+
    scala> testDF.createOrReplaceTempView("test")
    scala> // Do the transformation of the test data using the model and
predict the output values or lables. This is to compare the predicted value
and the actual label value
    scala> val tested = model.transform(testDF).select("features", "label",
"prediction")
    tested: org.apache.spark.sql.DataFrame = [features: vector, label:
double ... 1 more field]
    scala> tested.show()
      +--------------------+-----+-----------------+
      |            features|label|       prediction|
      +--------------------+-----+-----------------+
      |[7.4,0.7,0.0,1.9,...|  5.0|5.352730835898477|
      |[7.8,0.88,0.0,2.6...|  5.0|4.817999362011964|
      |[7.3,0.65,0.0,1.2...|  7.0|5.280106355653388|
      +--------------------+-----+-----------------+
    scala> // Prepare a dataset without the output/lables to predict the
output using the trained model
    scala> val predictDF = spark.sql("SELECT features FROM test")
    predictDF: org.apache.spark.sql.DataFrame = [features: vector]
    scala> predictDF.show()
      +--------------------+
      |            features|
      +--------------------+
      |[7.4,0.7,0.0,1.9,...|
      |[7.8,0.88,0.0,2.6...|
      |[7.3,0.65,0.0,1.2...|
      +--------------------+
    scala> // Do the transformation with the predict dataset and display
the predictions
    scala> val predicted = model.transform(predictDF).select("features",
```

```
    "prediction")
        predicted: org.apache.spark.sql.DataFrame = [features: vector,
prediction: double]
    scala> predicted.show()
    +--------------------+------------------+
    |            features|        prediction|
    +--------------------+------------------+
    |[7.4,0.7,0.0,1.9,...|5.352730835898477|
    |[7.8,0.88,0.0,2.6...|4.817999362011964|
    |[7.3,0.65,0.0,1.2...|5.280106355653388|
    +--------------------+------------------+
    scala> //IMPORTANT - To continue with the model persistence coming in
the next section, keep this session on.
```

The preceding code does a lot of things. It performs the following chain of activities in a pipeline:

1. It reads the wine data from the data file to form a training DataFrame.
2. Then it creates a `LinearRegression` object and sets the parameters.
3. It fits the model with the training data and this completes the estimator pipeline.
4. It creates a DataFrame containing test data. Typically, the test data will have both features and labels. This is to make sure that the model is right and used for comparing the predicted label and actual label.
5. Using the model created, it does a transformation with the test data, and from the DataFrame produced, extracts the features, input labels, and predictions. Note that while doing the transformation using the model, the labels are not required. In other words, the labels will not be used at all.
6. Using the model created, it does a transformation with the prediction data and from the DataFrame produced, extracts the features and predictions. Note that while doing the transformation using the model, the labels are not used. In other words, the labels are not used while doing the predictions. This completes a transformer pipeline.

> The pipelines in the preceding code snippet are single stage pipelines, and for this reason there is no need to use the Pipeline object. Multiple stage pipelines are going to be discussed in the following sections.

The fitting/testing phases are repeated iteratively in real-world use cases until the model is giving the desired results when doing predictions. Figure 2 elucidates the pipeline concept that is demonstrated through the code:
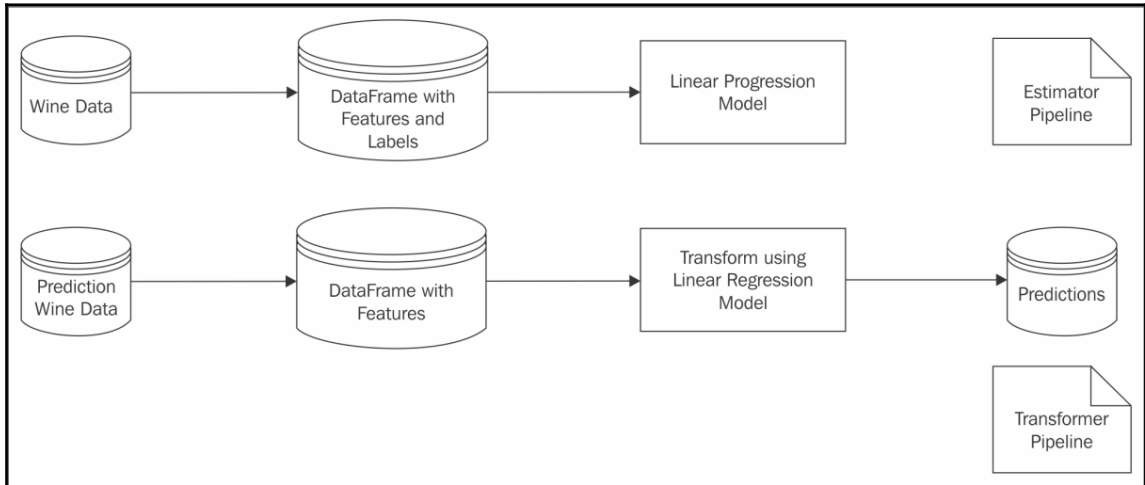
Figure 2

The following code demonstrates the same use case using Python. At the Python REPL prompt, try the following statements:

```
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.regression import LinearRegression
>>> from pyspark.ml.param import Param, Params
>>> from pyspark.sql import Row
>>> # TODO - Change this directory to the right location where the data
is stored
>>> dataDir = "/Users/RajT/Downloads/wine-quality/"
>>> # Create the the RDD by reading the wine data from the disk
>>> lines = sc.textFile(dataDir + "winequality-red.csv")
>>> splitLines = lines.map(lambda l: l.split(";"))
>>> # Vector is a data type with 0 based indices and double-typed
values. In that there are two types namely dense and sparse.
>>> # A dense vector is backed by a double array representing its entry
values
>>> # A sparse vector is backed by two parallel arrays: indices and
values
>>> wineDataRDD = splitLines.map(lambda p: (float(p[11]),
Vectors.dense([float(p[0]), float(p[1]), float(p[2]), float(p[3]),
float(p[4]), float(p[5]), float(p[6]), float(p[7]), float(p[8]),
float(p[9]), float(p[10])])))
>>> # Create the data frame containing the training data having two
```

```
columns. 1) The actula output or label of the data 2) The vector containing
the features
    >>> trainingDF = spark.createDataFrame(wineDataRDD, ['label',
'features'])
    >>> trainingDF.show()
      +-----+--------------------+
      |label|            features|
      +-----+--------------------+
      |  5.0|[7.4,0.7,0.0,1.9,...|
      |  5.0|[7.8,0.88,0.0,2.6...|
      |  5.0|[7.8,0.76,0.04,2....|
      |  6.0|[11.2,0.28,0.56,1...|
      |  5.0|[7.4,0.7,0.0,1.9,...|
      |  5.0|[7.4,0.66,0.0,1.8...|
      |  5.0|[7.9,0.6,0.06,1.6...|
      |  7.0|[7.3,0.65,0.0,1.2...|
      |  7.0|[7.8,0.58,0.02,2....|
      |  5.0|[7.5,0.5,0.36,6.1...|
      |  5.0|[6.7,0.58,0.08,1....|
      |  5.0|[7.5,0.5,0.36,6.1...|
      |  5.0|[5.6,0.615,0.0,1....|
      |  5.0|[7.8,0.61,0.29,1....|
      |  5.0|[8.9,0.62,0.18,3....|
      |  5.0|[8.9,0.62,0.19,3....|
      |  7.0|[8.5,0.28,0.56,1....|
      |  5.0|[8.1,0.56,0.28,1....|
      |  4.0|[7.4,0.59,0.08,4....|
      |  6.0|[7.9,0.32,0.51,1....|
      +-----+--------------------+
      only showing top 20 rows
    >>> # Create the object of the algorithm which is the Linear Regression
with the parameters
    >>> # Linear regression parameter to make lr.fit() use at most 10
iterations
    >>> lr = LinearRegression(maxIter=10)
    >>> # Create a trained model by fitting the parameters using the
training data
    >>> model = lr.fit(trainingDF)
    >>> # Once the model is prepared, to test the model, prepare the test
data containing the labels and feature vectors
    >>> testDF = spark.createDataFrame([(5.0, Vectors.dense([7.4, 0.7, 0.0,
1.9, 0.076, 25.0, 67.0, 0.9968, 3.2, 0.68,9.8])),(5.0,Vectors.dense([7.8,
0.88, 0.0, 2.6, 0.098, 11.0, 34.0, 0.9978, 3.51, 0.56, 9.4])),(7.0,
Vectors.dense([7.3, 0.65, 0.0, 1.2, 0.065, 15.0, 18.0, 0.9968, 3.36, 0.57,
9.5]))], ["label", "features"])
    >>> testDF.createOrReplaceTempView("test")
    >>> testDF.show()
      +-----+--------------------+
```

```
      |label|          features|
      +-----+------------------+
      |  5.0|[7.4,0.7,0.0,1.9,...|
      |  5.0|[7.8,0.88,0.0,2.6...|
      |  7.0|[7.3,0.65,0.0,1.2...|
      +-----+------------------+
>>> # Do the transformation of the test data using the model and
predict the output values or lables. This is to compare the predicted value
and the actual label value
>>> testTransform = model.transform(testDF)
>>> tested = testTransform.select("features", "label", "prediction")
>>> tested.show()
      +------------------+-----+----------------+
      |          features|label|      prediction|
      +------------------+-----+----------------+
      |[7.4,0.7,0.0,1.9,...|  5.0|5.352730835898477|
      |[7.8,0.88,0.0,2.6...|  5.0|4.817999362011964|
      |[7.3,0.65,0.0,1.2...|  7.0|5.280106355653388|
      +------------------+-----+----------------+
>>> # Prepare a dataset without the output/lables to predict the output
using the trained model
>>> predictDF = spark.sql("SELECT features FROM test")
>>> predictDF.show()
      +------------------+
      |          features|
      +------------------+
      |[7.4,0.7,0.0,1.9,...|
      |[7.8,0.88,0.0,2.6...|
      |[7.3,0.65,0.0,1.2...|
      +------------------+
>>> # Do the transformation with the predict dataset and display the
predictions
>>> predictTransform = model.transform(predictDF)
>>> predicted = predictTransform.select("features", "prediction")
>>> predicted.show()
      +------------------+----------------+
      |          features|      prediction|
      +------------------+----------------+
      |[7.4,0.7,0.0,1.9,...|5.352730835898477|
      |[7.8,0.88,0.0,2.6...|4.817999362011964|
      |[7.3,0.65,0.0,1.2...|5.280106355653388|
      +------------------+----------------+
>>> #IMPORTANT – To continue with the model persistence coming in the
next section, keep this session on.
```

As mentioned earlier, linear regression is statistical model and an approach for modeling the relationship between two types of variable. One is an independent variable, and the other is a dependent variable. The dependent variable is computed from the independent variables. In many cases, if there is only one independent variable, then the regression will be a simple linear regression. But in reality, in practical real-world use cases, there will be multitude of independent variables, just like in the wine dataset. This falls into the case of multiple linear regressions. This should not be confused with multivariate linear regression. In multivariate regression, multiple and correlated dependent variables are predicted.

In the use case that is being discussed here, the prediction is only done for one variable, which is the quality of the wine and hence it is a multiple linear regression and not a multivariate linear regression problem. Some schools even use multiple linear regression as univariate linear regression. In other words, irrespective of the number of independent variables, if there is only one dependent variable, it is termed as univariate linear regression.

Model persistence

Spark 2.0 comes with the ability to save and load machine learning models across programming languages with ease. In other words, you can create a machine learning model in Scala and load it in Python. This allows us to create a model in one system, save it, copy it, and use it in other systems. Continuing with the same Scala REPL prompt, try the following statements:

```
scala> // Assuming that the model definition line "val model =
lr.fit(trainingDF)" is still in context
scala> import org.apache.spark.ml.regression.LinearRegressionModel
  import org.apache.spark.ml.regression.LinearRegressionModel
scala> model.save("wineLRModelPath")
scala> val newModel = LinearRegressionModel.load("wineLRModelPath")
  newModel: org.apache.spark.ml.regression.LinearRegressionModel =
  linReg_6a880215ab96
```

Now the loaded model can be used for testing or prediction, just like the original model. Continuing with the same Python REPL prompt, try the following statements to load the model saved using the Scala program:

```
>>> from pyspark.ml.regression import LinearRegressionModel
>>> newModel = LinearRegressionModel.load("wineLRModelPath")
>>> newPredictTransform = newModel.transform(predictDF)
>>> newPredicted = newPredictTransform.select("features", "prediction")
>>> newPredicted.show()
  +-------------------+----------------+
```

```
|           features|       prediction|
+-------------------+-----------------+
|[7.4,0.7,0.0,1.9,...|5.352730835898477|
|[7.8,0.88,0.0,2.6...|4.817999362011964|
|[7.3,0.65,0.0,1.2...|5.280106355653388|
+-------------------+-----------------+
```

Wine classification

The dataset containing various features of white wine is used in this wine quality classification use case. The following are the features of the dataset:

- Fixed acidity
- Volatile acidity
- Citric acid
- Residual sugar
- Chlorides
- Free sulfur dioxide
- Total sulfur dioxide
- Density
- pH
- Sulphates
- Alcohol

Based on these features, the quality (score between 0 and 10) is determined. If the quality is less than 7, then it is classified as bad and a value of 0 is assigned to the label. If the quality is 7 or above, then it is classified as good and a value of 1 is assigned to the label. In other words, the classification value is the label of this dataset. Using this dataset, a model is going to be trained and then using the trained model, testing is done and predictions are made. This is a classification problem. The Logistic Regression algorithm is used to train the model. In this machine learning application use case, it deals with modeling the relationship between a dependent variable (wine quality) and a set of independent variables (the features of the wine).

At the Scala REPL prompt, try the following statements:

```scala
scala> import org.apache.spark.ml.classification.LogisticRegression
  import org.apache.spark.ml.classification.LogisticRegression
scala> import org.apache.spark.ml.param.ParamMap
  import org.apache.spark.ml.param.ParamMap
scala> import org.apache.spark.ml.linalg.{Vector, Vectors}
  import org.apache.spark.ml.linalg.{Vector, Vectors}
```

```
scala> import org.apache.spark.sql.Row
   import org.apache.spark.sql.Row
scala> // TODO - Change this directory to the right location where the
data is stored
scala> val dataDir = "/Users/RajT/Downloads/wine-quality/"
   dataDir: String = /Users/RajT/Downloads/wine-quality/
scala> // Define the case class that holds the wine data
scala> case class Wine(FixedAcidity: Double, VolatileAcidity: Double,
CitricAcid: Double, ResidualSugar: Double, Chlorides: Double,
FreeSulfurDioxide: Double, TotalSulfurDioxide: Double, Density: Double, PH:
Double, Sulphates: Double, Alcohol: Double, Quality: Double)
   defined class Wine
scala> // Create the the RDD by reading the wine data from the disk
scala> val wineDataRDD = sc.textFile(dataDir + "winequality-
white.csv").map(_.split(";")).map(w => Wine(w(0).toDouble, w(1).toDouble,
w(2).toDouble, w(3).toDouble, w(4).toDouble, w(5).toDouble, w(6).toDouble,
w(7).toDouble, w(8).toDouble, w(9).toDouble, w(10).toDouble,
w(11).toDouble))
   wineDataRDD: org.apache.spark.rdd.RDD[Wine] = MapPartitionsRDD[35] at
map at <console>:36
scala> // Create the data frame containing the training data having two
columns. 1) The actula output or label of the data 2) The vector containing
the features
scala> val trainingDF = wineDataRDD.map(w => (if(w.Quality < 7) 0D else
1D, Vectors.dense(w.FixedAcidity, w.VolatileAcidity, w.CitricAcid,
w.ResidualSugar, w.Chlorides, w.FreeSulfurDioxide, w.TotalSulfurDioxide,
w.Density, w.PH, w.Sulphates, w.Alcohol))).toDF("label", "features")
   trainingDF: org.apache.spark.sql.DataFrame = [label: double,
features: vector]
scala> trainingDF.show()
   +-----+--------------------+
   |label|            features|
   +-----+--------------------+
   |  0.0|[7.0,0.27,0.36,20...|
   |  0.0|[6.3,0.3,0.34,1.6...|
   |  0.0|[8.1,0.28,0.4,6.9...|
   |  0.0|[7.2,0.23,0.32,8....|
   |  0.0|[7.2,0.23,0.32,8....|
   |  0.0|[8.1,0.28,0.4,6.9...|
   |  0.0|[6.2,0.32,0.16,7....|
   |  0.0|[7.0,0.27,0.36,20...|
   |  0.0|[6.3,0.3,0.34,1.6...|
   |  0.0|[8.1,0.22,0.43,1....|
   |  0.0|[8.1,0.27,0.41,1....|
   |  0.0|[8.6,0.23,0.4,4.2...|
   |  0.0|[7.9,0.18,0.37,1....|
   |  1.0|[6.6,0.16,0.4,1.5...|
   |  0.0|[8.3,0.42,0.62,19...|
```

```
        |  1.0|[6.6,0.17,0.38,1....|
        |  0.0|[6.3,0.48,0.04,1....|
        |  1.0|[6.2,0.66,0.48,1....|
        |  0.0|[7.4,0.34,0.42,1....|
        |  0.0|[6.5,0.31,0.14,7....|
        +-----+--------------------+
        only showing top 20 rows
    scala> // Create the object of the algorithm which is the Logistic
Regression
    scala> val lr = new LogisticRegression()
      lr: org.apache.spark.ml.classification.LogisticRegression =
logreg_a7e219daf3e1
    scala> // LogisticRegression parameter to make lr.fit() use at most 10
iterations and the regularization parameter.
    scala> // When a higher degree polynomial used by the algorithm to fit
a set of points in a linear regression model, to prevent overfitting,
regularization is used and this parameter is just for that
    scala> lr.setMaxIter(10).setRegParam(0.01)
      res8: lr.type = logreg_a7e219daf3e1
    scala> // Create a trained model by fitting the parameters using the
training data
    scala> val model = lr.fit(trainingDF)
      model: org.apache.spark.ml.classification.LogisticRegressionModel =
logreg_a7e219daf3e1
    scala> // Once the model is prepared, to test the model, prepare the
test data containing the labels and feature vectors
    scala> val testDF = spark.createDataFrame(Seq((1.0,
Vectors.dense(6.1,0.32,0.24,1.5,0.036,43,140,0.9894,3.36,0.64,10.7)),(0.0,
Vectors.dense(5.2,0.44,0.04,1.4,0.036,38,124,0.9898,3.29,0.42,12.4)),(0.0,
Vectors.dense(7.2,0.32,0.47,5.1,0.044,19,65,0.9951,3.38,0.36,9)),(0.0,Vecto
rs.dense(6.4,0.595,0.14,5.2,0.058,15,97,0.991,3.03,0.41,12.6)))).toDF("labe
l", "features")
      testDF: org.apache.spark.sql.DataFrame = [label: double, features:
vector]
    scala> testDF.show()
        +-----+--------------------+
        |label|            features|
        +-----+--------------------+
        |  1.0|[6.1,0.32,0.24,1....|
        |  0.0|[5.2,0.44,0.04,1....|
        |  0.0|[7.2,0.32,0.47,5....|
        |  0.0|[6.4,0.595,0.14,5...|
        +-----+--------------------+
    scala> testDF.createOrReplaceTempView("test")
    scala> // Do the transformation of the test data using the model and
predict the output values or labels. This is to compare the predicted value
and the actual label value
    scala> val tested = model.transform(testDF).select("features", "label",
```

```
      "prediction")
      tested: org.apache.spark.sql.DataFrame = [features: vector, label:
double ... 1 more field]
      scala> tested.show()
        +--------------------+-----+----------+
        |            features|label|prediction|
        +--------------------+-----+----------+
        |[6.1,0.32,0.24,1....|  1.0|       0.0|
        |[5.2,0.44,0.04,1....|  0.0|       0.0|
        |[7.2,0.32,0.47,5....|  0.0|       0.0|
        |[6.4,0.595,0.14,5...|  0.0|       0.0|
        +--------------------+-----+----------+
      scala> // Prepare a dataset without the output/lables to predict the
output using the trained model
      scala> val predictDF = spark.sql("SELECT features FROM test")
       predictDF: org.apache.spark.sql.DataFrame = [features: vector]
      scala> predictDF.show()
        +--------------------+
        |            features|
        +--------------------+
        |[6.1,0.32,0.24,1....|
        |[5.2,0.44,0.04,1....|
        |[7.2,0.32,0.47,5....|
        |[6.4,0.595,0.14,5...|
        +--------------------+
      scala> // Do the transformation with the predict dataset and display
the predictions
      scala> val predicted = model.transform(predictDF).select("features",
"prediction")
      predicted: org.apache.spark.sql.DataFrame = [features: vector,
prediction: double]
      scala> predicted.show()
        +--------------------+----------+
        |            features|prediction|
        +--------------------+----------+
        |[6.1,0.32,0.24,1....|       0.0|
        |[5.2,0.44,0.04,1....|       0.0|
        |[7.2,0.32,0.47,5....|       0.0|
        |[6.4,0.595,0.14,5...|       0.0|
        +--------------------+----------+
```

The preceding code snippet works exactly like a linear regression use case, except for the model used here. The model used here is Logistic Regression and its label takes only two values, 0 and 1. Creating the model, testing the model, and then the predictions are all similar here. In other words, the pipelines look very similar.

The following code demonstrates the same use case using Python. At the Python REPL prompt, try the following statements:

```
>>> from pyspark.ml.linalg import Vectors
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.param import Param, Params
>>> from pyspark.sql import Row
>>> # TODO - Change this directory to the right location where the data
is stored
>>> dataDir = "/Users/RajT/Downloads/wine-quality/"
>>> # Create the the RDD by reading the wine data from the disk
>>> lines = sc.textFile(dataDir + "winequality-white.csv")
>>> splitLines = lines.map(lambda l: l.split(";"))
>>> wineDataRDD = splitLines.map(lambda p: (float(0) if (float(p[11]) <
7) else float(1), Vectors.dense([float(p[0]), float(p[1]), float(p[2]),
float(p[3]), float(p[4]), float(p[5]), float(p[6]), float(p[7]),
float(p[8]), float(p[9]), float(p[10])])))
>>> # Create the data frame containing the training data having two
columns. 1) The actula output or label of the data 2) The vector containing
the features
>>> trainingDF = spark.createDataFrame(wineDataRDD, ['label',
'features'])
>>> trainingDF.show()
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|[7.0,0.27,0.36,20...|
|  0.0|[6.3,0.3,0.34,1.6...|
|  0.0|[8.1,0.28,0.4,6.9...|
|  0.0|[7.2,0.23,0.32,8....|
|  0.0|[7.2,0.23,0.32,8....|
|  0.0|[8.1,0.28,0.4,6.9...|
|  0.0|[6.2,0.32,0.16,7....|
|  0.0|[7.0,0.27,0.36,20...|
|  0.0|[6.3,0.3,0.34,1.6...|
|  0.0|[8.1,0.22,0.43,1....|
|  0.0|[8.1,0.27,0.41,1....|
|  0.0|[8.6,0.23,0.4,4.2...|
|  0.0|[7.9,0.18,0.37,1....|
|  1.0|[6.6,0.16,0.4,1.5...|
|  0.0|[8.3,0.42,0.62,19...|
|  1.0|[6.6,0.17,0.38,1....|
|  0.0|[6.3,0.48,0.04,1....|
|  1.0|[6.2,0.66,0.48,1....|
|  0.0|[7.4,0.34,0.42,1....|
|  0.0|[6.5,0.31,0.14,7....|
+-----+--------------------+
only showing top 20 rows
```

```
    >>> # Create the object of the algorithm which is the Logistic
Regression with the parameters
    >>> # LogisticRegression parameter to make lr.fit() use at most 10
iterations and the regularization parameter.
    >>> # When a higher degree polynomial used by the algorithm to fit a
set of points in a linear regression model, to prevent overfitting,
regularization is used and this parameter is just for that
    >>> lr = LogisticRegression(maxIter=10, regParam=0.01)
    >>> # Create a trained model by fitting the parameters using the
training data
    >>> model = lr.fit(trainingDF)
    >>> # Once the model is prepared, to test the model, prepare the test
data containing the labels and feature vectors
    >>> testDF = spark.createDataFrame([(1.0,
Vectors.dense([6.1,0.32,0.24,1.5,0.036,43,140,0.9894,3.36,0.64,10.7])),(0.0
,
Vectors.dense([5.2,0.44,0.04,1.4,0.036,38,124,0.9898,3.29,0.42,12.4])),(0.0
, Vectors.dense([7.2,0.32,0.47,5.1,0.044,19,65,0.9951,3.38,0.36,9])),(0.0,
Vectors.dense([6.4,0.595,0.14,5.2,0.058,15,97,0.991,3.03,0.41,12.6]))],
["label", "features"])
    >>> testDF.createOrReplaceTempView("test")
    >>> testDF.show()
      +-----+--------------------+
      |label|            features|
      +-----+--------------------+
      |  1.0|[6.1,0.32,0.24,1....|
      |  0.0|[5.2,0.44,0.04,1....|
      |  0.0|[7.2,0.32,0.47,5....|
      |  0.0|[6.4,0.595,0.14,5...|
      +-----+--------------------+
    >>> # Do the transformation of the test data using the model and
predict the output values or lables. This is to compare the predicted value
and the actual label value
    >>> testTransform = model.transform(testDF)
    >>> tested = testTransform.select("features", "label", "prediction")
    >>> tested.show()
      +--------------------+-----+----------+
      |            features|label|prediction|
      +--------------------+-----+----------+
      |[6.1,0.32,0.24,1....|  1.0|       0.0|
      |[5.2,0.44,0.04,1....|  0.0|       0.0|
      |[7.2,0.32,0.47,5....|  0.0|       0.0|
      |[6.4,0.595,0.14,5...|  0.0|       0.0|
      +--------------------+-----+----------+
    >>> # Prepare a dataset without the output/lables to predict the output
using the trained model
    >>> predictDF = spark.sql("SELECT features FROM test")
    >>> predictDF.show()
```

```
+-------------------+
|           features|
+-------------------+
|[6.1,0.32,0.24,1....|
|[5.2,0.44,0.04,1....|
|[7.2,0.32,0.47,5....|
|[6.4,0.595,0.14,5...|
+-------------------+
>>> # Do the transformation with the predict dataset and display the
predictions
>>> predictTransform = model.transform(predictDF)
>>> predicted = testTransform.select("features", "prediction")
>>> predicted.show()
+-------------------+----------+
|           features|prediction|
+-------------------+----------+
|[6.1,0.32,0.24,1....|       0.0|
|[5.2,0.44,0.04,1....|       0.0|
|[7.2,0.32,0.47,5....|       0.0|
|[6.4,0.595,0.14,5...|       0.0|
+-------------------+----------+
```

Logistic regression is very similar to linear regression. The major difference in Logistic regression is that its dependent variable is a categorical variable. In other words, the dependent variable takes only a selected set of values. In this use case the values are 0 or 1. The value 0 means that the wine quality is bad and the value 1 means that the wine quality is good. To be more precise, here, the dependent variable used is a binary dependent variable.

The use cases covered so far have only a handful of features. But in real-world use cases the number of features is going to be really huge, especially in machine learning use cases where lots of text processing is done. The next section is going to discuss one such use case.

Spam filtering

Spam filtering is a very common use case that is used in many applications. It is ubiquitous in e-mail applications. It is one of the most widely used classification problems. In a typical mail server, a huge number of e-mails are processed. The spam filtering is done on the e-mails received before they are delivered to the recipient's mailboxes. For any machine learning algorithm, a model has to be trained before making a prediction. To train the model, training data is required. How is training data collected? A trivial way is that the users themselves mark some of the e-mails received as spam. Use all the e-mails in the mail server as training data and keep refreshing the model on a regular basis. This includes spam

and non-spam e-mails. When the model has a good sample of both kinds of e-mail, the prediction is going to be good.

The spam filtering use case covered here is not a full-blown production-ready application, but it gives a good insight into how one can be built. Here, instead of using the entire text of an e-mail, only one line is used for simplicity. If this is to be extended to process real e-mails, instead of the single strings, read the contents of a full e-mail to one string and proceed as per the logic given in this application.

Unlike the numeric features that are covered in the earlier use cases of this chapter, the input here is pure text and selecting features is not as easy as those use cases. The lines are split into words to have a bag of words and the words are chosen as features. Since it is easy to process numerical features, these words are transformed to hashed term frequency vectors. In other words, the series of words or terms in the lines are converted to their term frequencies using a hashing method. So even in small-scale text processing use cases, there will be thousands of features. That is why they need to be hashed for easy comparison.

As discussed earlier, in typical machine learning applications, the input data needs to undergo lots of pre-processing to get it in the right form of features and labels in order to build the model. This typically forms a pipeline of transformations and estimations. In this use case, the incoming lines are split into words, and those words are transformed using HashingTF algorithm and then a LogisticRegression model is trained before doing the prediction. This is done using the Pipeline abstraction in the Spark machine learning library. At the Scala REPL prompt, try the following statements:

```
scala> import org.apache.spark.ml.classification.LogisticRegression
  import org.apache.spark.ml.classification.LogisticRegression
scala> import org.apache.spark.ml.param.ParamMap
  import org.apache.spark.ml.param.ParamMap
scala> import org.apache.spark.ml.linalg.{Vector, Vectors}
  import org.apache.spark.ml.linalg.{Vector, Vectors}
scala> import org.apache.spark.sql.Row
  import org.apache.spark.sql.Row
scala> import org.apache.spark.ml.Pipeline
  import org.apache.spark.ml.Pipeline
scala> import org.apache.spark.ml.feature.{HashingTF, Tokenizer,
RegexTokenizer, Word2Vec, StopWordsRemover}
  import org.apache.spark.ml.feature.{HashingTF, Tokenizer,
RegexTokenizer, Word2Vec, StopWordsRemover}
scala> // Prepare training documents from a list of messages from
emails used to filter them as spam or not spam
scala> // If the original message is a spam then the label is 1 and if
the message is genuine then the label is 0
scala> val training = spark.createDataFrame(Seq(("you@example.com",
"hope you are well", 0.0),("raj@example.com", "nice to hear from you",
```

```
0.0),("thomas@example.com", "happy holidays", 0.0),("mark@example.com",
"see you tomorrow", 0.0),("xyz@example.com", "save money",
1.0),("top10@example.com", "low interest rate",
1.0),("marketing@example.com", "cheap loan", 1.0))).toDF("email",
"message", "label")
    training: org.apache.spark.sql.DataFrame = [email: string, message:
string ... 1 more field]
  scala> training.show()
    +-------------------+--------------------+-----+
    |              email|             message|label|
    +-------------------+--------------------+-----+
    |      you@example.com|    hope you are well|  0.0|
    |      raj@example.com|nice to hear from...|  0.0|
    |   thomas@example.com|       happy holidays|  0.0|
    |     mark@example.com|     see you tomorrow|  0.0|
    |      xyz@example.com|          save money|  1.0|
    |    top10@example.com|    low interest rate|  1.0|
    |marketing@example...|          cheap loan|  1.0|
    +-------------------+--------------------+-----+
  scala>  // Configure an Spark machine learning pipeline, consisting of
three stages: tokenizer, hashingTF, and lr.
  scala> val tokenizer = new
Tokenizer().setInputCol("message").setOutputCol("words")
    tokenizer: org.apache.spark.ml.feature.Tokenizer = tok_166809bf629c
  scala> val hashingTF = new
HashingTF().setNumFeatures(1000).setInputCol("words").setOutputCol("feature
s")
    hashingTF: org.apache.spark.ml.feature.HashingTF =
hashingTF_e43616e13d19
  scala> // LogisticRegression parameter to make lr.fit() use at most 10
iterations and the regularization parameter.
  scala> // When a higher degree polynomial used by the algorithm to fit
a set of points in a linear regression model, to prevent overfitting,
regularization is used and this parameter is just for that
  scala> val lr = new
LogisticRegression().setMaxIter(10).setRegParam(0.01)
    lr: org.apache.spark.ml.classification.LogisticRegression =
logreg_ef3042fc75a3
  scala> val pipeline = new Pipeline().setStages(Array(tokenizer,
hashingTF, lr))
    pipeline: org.apache.spark.ml.Pipeline = pipeline_658b5edef0f2
  scala> // Fit the pipeline to train the model to study the messages
  scala> val model = pipeline.fit(training)
    model: org.apache.spark.ml.PipelineModel = pipeline_658b5edef0f2
  scala> // Prepare messages for prediction, which are not categorized
and leaving upto the algorithm to predict
  scala> val test = spark.createDataFrame(Seq(("you@example.com", "how
are you"),("jain@example.com", "hope doing well"),("caren@example.com",
```

```
    "want some money"),("zhou@example.com", "secure
loan"),("ted@example.com","need loan"))).toDF("email", "message")
    test: org.apache.spark.sql.DataFrame = [email: string, message:
string]
    scala> test.show()
    +----------------+--------------+
    |           email|       message|
    +----------------+--------------+
    |  you@example.com|   how are you|
    | jain@example.com|hope doing well|
    |caren@example.com|want some money|
    | zhou@example.com|    secure loan|
    |  ted@example.com|      need loan|
    +----------------+--------------+
    scala> // Make predictions on the new messages
    scala> val prediction = model.transform(test).select("email",
"message", "prediction")
    prediction: org.apache.spark.sql.DataFrame = [email: string, message:
string ... 1 more field]
    scala> prediction.show()
    +----------------+--------------+----------+
    |           email|       message|prediction|
    +----------------+--------------+----------+
    |  you@example.com|   how are you|       0.0|
    | jain@example.com|hope doing well|       0.0|
    |caren@example.com|want some money|       1.0|
    | zhou@example.com|    secure loan|       1.0|
    |  ted@example.com|      need loan|       1.0|
    +----------------+--------------+----------+
```

The preceding code snippet does the typical chain of activities: preparing training data, creating the model using the Pipeline abstraction, and then predicting using the test data. It doesn't reveal how the features are created and processed. In an application development perspective, Spark machine learning library does the heavy lifting and does everything under the hood using the Pipeline abstraction. If the Pipeline methodology is not used, then tokenisation and then hashing are to be done as a separate DataFrame transformation. The following code snippet executed as the continuation of the preceding commands will give an insight into how that can be done as simple transformations to see the features using the naked eyes:

```
    scala> val wordsDF = tokenizer.transform(training)
    wordsDF: org.apache.spark.sql.DataFrame = [email: string, message:
string ... 2 more fields]
    scala> wordsDF.createOrReplaceTempView("word")
    scala> val selectedFieldstDF = spark.sql("SELECT message, words FROM
word")
    selectedFieldstDF: org.apache.spark.sql.DataFrame = [message: string,
```

```
words: array<string>]
    scala> selectedFieldstDF.show()
      +-------------------+-------------------+
      |            message|              words|
      +-------------------+-------------------+
      |    hope you are well|[hope, you, are, ...|
      |nice to hear from...|[nice, to, hear, ...|
      |      happy holidays|   [happy, holidays]|
      |    see you tomorrow|[see, you, tomorrow]|
      |          save money|      [save, money]|
      |    low interest rate|[low, interest, r...|
      |          cheap loan|      [cheap, loan]|
      +-------------------+-------------------+
    scala> val featurizedDF = hashingTF.transform(wordsDF)
      featurizedDF: org.apache.spark.sql.DataFrame = [email: string,
message: string ... 3 more fields]
    scala> featurizedDF.createOrReplaceTempView("featurized")
    scala> val selectedFeaturizedFieldstDF = spark.sql("SELECT words,
features FROM featurized")
      selectedFeaturizedFieldstDF: org.apache.spark.sql.DataFrame = [words:
array<string>, features: vector]
    scala> selectedFeaturizedFieldstDF.show()
      +-------------------+-------------------+
      |              words|           features|
      +-------------------+-------------------+
      |[hope, you, are, ...|(1000,[0,138,157,...|
      |[nice, to, hear, ...|(1000,[370,388,42...|
      |   [happy, holidays]|(1000,[141,457],[...|
      |[see, you, tomorrow]|(1000,[25,425,515...|
      |      [save, money]|(1000,[242,520],[...|
      |[low, interest, r...|(1000,[70,253,618...|
      |      [cheap, loan]|(1000,[410,666],[...|
      +-------------------+-------------------+
```

The same use case implemented in Python is as follows. At the Python REPL prompt, try
the following statements:

```
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.classification import LogisticRegression
>>> from pyspark.ml.feature import HashingTF, Tokenizer
>>> from pyspark.sql import Row
>>> # Prepare training documents from a list of messages from emails
used to filter them as spam or not spam
>>> # If the original message is a spam then the label is 1 and if the
message is genuine then the label is 0
>>> LabeledDocument = Row("email", "message", "label")
>>> training = spark.createDataFrame([("you@example.com", "hope you are
well", 0.0),("raj@example.com", "nice to hear from you",
```

```
0.0),("thomas@example.com", "happy holidays", 0.0),("mark@example.com",
"see you tomorrow", 0.0),("xyz@example.com", "save money",
1.0),("top10@example.com", "low interest rate",
1.0),("marketing@example.com", "cheap loan", 1.0)], ["email", "message",
"label"])
    >>> training.show()
    +-------------------+-------------------+-----+
    |              email|            message|label|
    +-------------------+-------------------+-----+
    |     you@example.com|   hope you are well|  0.0|
    |     raj@example.com|nice to hear from...|  0.0|
    |  thomas@example.com|      happy holidays|  0.0|
    |    mark@example.com|    see you tomorrow|  0.0|
    |     xyz@example.com|          save money|  1.0|
    |   top10@example.com|   low interest rate|  1.0|
    |marketing@example...|          cheap loan|  1.0|
    +-------------------+-------------------+-----+
    >>> # Configure an Spark machin learning pipeline, consisting of three
stages: tokenizer, hashingTF, and lr.
    >>> tokenizer = Tokenizer(inputCol="message", outputCol="words")
    >>> hashingTF = HashingTF(inputCol="words", outputCol="features")
    >>> # LogisticRegression parameter to make lr.fit() use at most 10
iterations and the regularization parameter.
    >>> # When a higher degree polynomial used by the algorithm to fit a
set of points in a linear regression model, to prevent overfitting,
regularization is used and this parameter is just for that
    >>> lr = LogisticRegression(maxIter=10, regParam=0.01)
    >>> pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
    >>> # Fit the pipeline to train the model to study the messages
    >>> model = pipeline.fit(training)
    >>> # Prepare messages for prediction, which are not categorized and
leaving upto the algorithm to predict
    >>> test = spark.createDataFrame([("you@example.com", "how are
you"),("jain@example.com", "hope doing well"),("caren@example.com", "want
some money"),("zhou@example.com", "secure loan"),("ted@example.com","need
loan")], ["email", "message"])
    >>> test.show()
    +----------------+---------------+
    |           email|        message|
    +----------------+---------------+
    |  you@example.com|    how are you|
    | jain@example.com|hope doing well|
    |caren@example.com|want some money|
    | zhou@example.com|    secure loan|
    |  ted@example.com|      need loan|
    +----------------+---------------+
    >>> # Make predictions on the new messages
    >>> prediction = model.transform(test).select("email", "message",
```

```
    "prediction")
    >>> prediction.show()
    +----------------+--------------+----------+
    |           email|       message|prediction|
    +----------------+--------------+----------+
    |  you@example.com|   how are you|       0.0|
    | jain@example.com|hope doing well|      0.0|
    |caren@example.com|want some money|      1.0|
    | zhou@example.com|    secure loan|      1.0|
    |  ted@example.com|      need loan|      1.0|
    +----------------+--------------+----------+
```

As discussed in earlier, the transformations abstracted by the Pipeline is elucidated as follows using Python explicitly. The following code snippet executed as the continuation of the preceding commands will give an insight into how that can be done as simple transformations to see the features using the naked eyes:

```
>>> wordsDF = tokenizer.transform(training)
>>> wordsDF.createOrReplaceTempView("word")
>>> selectedFieldstDF = spark.sql("SELECT message, words FROM word")
>>> selectedFieldstDF.show()
    +-------------------+--------------------+
    |            message|               words|
    +-------------------+--------------------+
    |    hope you are well|[hope, you, are, ...|
    |nice to hear from...|[nice, to, hear, ...|
    |        happy holidays|   [happy, holidays]|
    |     see you tomorrow|[see, you, tomorrow]|
    |           save money|        [save, money]|
    |    low interest rate|[low, interest, r...|
    |           cheap loan|        [cheap, loan]|
    +-------------------+--------------------+
>>> featurizedDF = hashingTF.transform(wordsDF)
>>> featurizedDF.createOrReplaceTempView("featurized")
>>> selectedFeaturizedFieldstDF = spark.sql("SELECT words, features
FROM featurized")
>>> selectedFeaturizedFieldstDF.show()
    +-------------------+--------------------+
    |              words|            features|
    +-------------------+--------------------+
    |[hope, you, are, ...|(262144,[128160,1...|
    |[nice, to, hear, ...|(262144,[22346,10...|
    |   [happy, holidays]|(262144,[86293,23...|
    |[see, you, tomorrow]|(262144,[29129,21...|
    |        [save, money]|(262144,[199496,2...|
    |[low, interest, r...|(262144,[68685,13...|
    |        [cheap, loan]|(262144,[12946,16...|
    +-------------------+--------------------+
```

Based on the insight provided in the preceding use case, lots of text processing machine learning applications can be developed by abstracting away lots of transformations using the Spark machine learning library Pipelines.

> **TIP**
>
> Just like the way the machine learning models are persisted to the media, all of the Spark machine learning library Pipelines can also be persisted to the media and reloaded by other programs.

Feature algorithms

In real-world use cases, it is not very easy to get the raw data in the appropriate form of features and labels in order to train the model. Doing lots of pre-processing is very common. Unlike other data processing paradigms, Spark in conjunction with the Spark machine learning library provides a comprehensive set of tools and algorithms for this purpose. This pre-processing algorithms can be put into three categories:

- Feature extraction
- Feature transformation
- Feature selection

The process of extracting the features from the raw data is feature extraction. The HashingTF that was used in the preceding use case is a good example of an algorithm that converts terms of text data to feature vectors. The process of transforming features into different formats is feature transformation. The process of selecting a subset of features from a super set is feature selection. Covering all these is beyond the scope of this chapter, but the next section is going to discuss an Estimator, which is an algorithm that is used to extract features, that is used to find synonyms of words in documents,. These are not the word's actual synonyms, but the words that are related to a given word in a context.

Finding synonyms

A synonym is a word or phrase that has exactly the same meaning or very close meaning to another word. In a purely literature perspective this explanation is correct, but in a much wider perspective, in a given context, some of the words will have a very close relationship, and that is also called synonymous in this context. For example, Roger Federer is *synonymous* with Tennis. Finding this kind of synonym in context is a very common requirement in entity recognition, machine translation, and so on. The **Word2Vec** algorithm computes a distributed vector representation of words from the words of a given document or collection of words. If this vector space is taken, the words that have similarity or synonymity will be close to each other.

The University of California Irvine Machine Learning Repository (`http://archive.ics.uci.edu/ml/index.html`) provides a lot of datasets as a service to those who are interested to learn machine learning. The Twenty Newsgroups Dataset (`http://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups`) is being used here to find synonyms of words in context. It contains a dataset consists of 20,000 messages taken from 20 newsgroups.

> The Twenty Newsgroups Dataset download link lets you download the dataset discussed here. The file `20_newsgroups.tar.gz` is to be downloaded and unzipped. The data directory used in the following code snippets should point to the directory where the data is available in unzipped form. If the Spark Driver is giving out of memory error because of the huge size of the data, remove some of the newsgroups data that is of not interest and experiment with a subset of the data. Here, to train the model, only the following news group data is used: talk.politics.guns, talk.politics.mideast, talk.politics.misc, and talk.religion.misc.

At the Scala REPL prompt, try the following statements:

```
scala> import org.apache.spark.ml.feature.{HashingTF, Tokenizer,
RegexTokenizer, Word2Vec, StopWordsRemover}
  import org.apache.spark.ml.feature.{HashingTF, Tokenizer,
RegexTokenizer, Word2Vec, StopWordsRemover}
scala> // TODO - Change this directory to the right location where the
data is stored
scala> val dataDir = "/Users/RajT/Downloads/20_newsgroups/*"
  dataDir: String = /Users/RajT/Downloads/20_newsgroups/*
scala> //Read the entire text into a DataFrame
scala> // Only the following directories under the data directory has
benn considered for running this program talk.politics.guns,
talk.politics.mideast, talk.politics.misc, talk.religion.misc. All other
directories have been removed before running this program. There is no harm
```

```
     in retaining all the data. The only difference will be in the output.
         scala>  val textDF = sc.wholeTextFiles(dataDir).map{case(file, text) =>
     text}.map(Tuple1.apply).toDF("sentence")
           textDF: org.apache.spark.sql.DataFrame = [sentence: string]
         scala>  // Tokenize the sentences to words
         scala>  val regexTokenizer = new
     RegexTokenizer().setInputCol("sentence").setOutputCol("words").setPattern("
     \\w+").setGaps(false)
           regexTokenizer: org.apache.spark.ml.feature.RegexTokenizer =
     regexTok_ba7ce8ec2333
         scala> val tokenizedDF = regexTokenizer.transform(textDF)
           tokenizedDF: org.apache.spark.sql.DataFrame = [sentence: string,
     words: array<string>]
         scala>  // Remove the stop words such as a, an the, I etc which doesn't
     have any specific relevance to the synonyms
         scala> val remover = new
     StopWordsRemover().setInputCol("words").setOutputCol("filtered")
           remover: org.apache.spark.ml.feature.StopWordsRemover =
     stopWords_775db995b8e8
         scala> //Remove the stop words from the text
         scala> val filteredDF = remover.transform(tokenizedDF)
           filteredDF: org.apache.spark.sql.DataFrame = [sentence: string,
     words: array<string> ... 1 more field]
         scala> //Prepare the Estimator
         scala> //It sets the vector size, and the method setMinCount sets the
     minimum number of times a token must appear to be included in the word2vec
     model's vocabulary.
         scala> val word2Vec = new
     Word2Vec().setInputCol("filtered").setOutputCol("result").setVectorSize(3).
     setMinCount(0)
           word2Vec: org.apache.spark.ml.feature.Word2Vec = w2v_bb03091c4439
         scala> //Train the model
         scala> val model = word2Vec.fit(filteredDF)
           model: org.apache.spark.ml.feature.Word2VecModel = w2v_bb03091c4439
         scala> //Find 10 synonyms of a given word
         scala> val synonyms1 = model.findSynonyms("gun", 10)
           synonyms1: org.apache.spark.sql.DataFrame = [word: string,
     similarity: double]
         scala> synonyms1.show()
         +---------+------------------+
         |     word|        similarity|
         +---------+------------------+
         |      twa|0.9999976163843671|
         |cigarette|0.9999943935045497|
         |    sorts|0.9999885527530025|
         |       jj|0.9999827967650881|
         |presently|0.9999792188771406|
         |    laden|0.9999775888361028|
```

```
|     notion|0.9999775296680583|
|   settlers|0.9999746245431419|
|motivated|0.9999694932468436|
|qualified|0.9999678135106314|
+---------+------------------+
scala> //Find 10 synonyms of a different word
scala> val synonyms2 = model.findSynonyms("crime", 10)
   synonyms2: org.apache.spark.sql.DataFrame = [word: string,
similarity: double]
scala> synonyms2.show()
+-----------+------------------+
|       word|        similarity|
+-----------+------------------+
| abominable|0.9999997331058447|
|authorities|0.9999946968941679|
|cooperation|0.9999892536435327|
|   mortazavi| 0.999986396931714|
|herzegovina|0.9999861828226779|
|  important|0.9999853354260315|
|       1950s|0.9999832312575262|
|     analogy|0.9999828272311249|
|        bits|0.9999820987679822|
|technically|0.9999808208936487|
+-----------+------------------+
```

The preceding code snippet is loaded with a lot of functionality. The dataset is read from the filesystem into a DataFrame as one sentence of text from a given file. Then tokenisation is done to convert the sentences into words using regular expressions and removing the gaps. Then, from those words, the stop words are removed so that we only have relevant words. Finally, using the **Word2Vec** estimator, a model is trained with the data prepared. From the trained model, synonyms are determined.

The following code demonstrates the same use case using Python. At the Python REPL prompt, try the following statements:

```
>>> from pyspark.ml.feature import Word2Vec
>>> from pyspark.ml.feature import RegexTokenizer
>>> from pyspark.sql import Row
>>> # TODO – Change this directory to the right location where the data
is stored
>>> dataDir = "/Users/RajT/Downloads/20_newsgroups/*"
>>> # Read the entire text into a DataFrame. Only the following
directories under the data directory has benn considered for running this
program talk.politics.guns, talk.politics.mideast, talk.politics.misc,
talk.religion.misc. All other directories have been removed before running
this program. There is no harm in retaining all the data. The only
difference will be in the output.
```

```
>>> textRDD = sc.wholeTextFiles(dataDir).map(lambda recs:
Row(sentence=recs[1]))
>>> textDF = spark.createDataFrame(textRDD)
>>> # Tokenize the sentences to words
>>> regexTokenizer = RegexTokenizer(inputCol="sentence",
outputCol="words", gaps=False, pattern="\\w+")
>>> tokenizedDF = regexTokenizer.transform(textDF)
>>> # Prepare the Estimator
>>> # It sets the vector size, and the parameter minCount sets the
minimum number of times a token must appear to be included in the word2vec
model's vocabulary.
>>> word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="words",
outputCol="result")
>>> # Train the model
>>> model = word2Vec.fit(tokenizedDF)
>>> # Find 10 synonyms of a given word
>>> synonyms1 = model.findSynonyms("gun", 10)
>>> synonyms1.show()
  +---------+------------------+
  |     word|        similarity|
  +---------+------------------+
  | strapped|0.9999918504219028|
  |    bingo|0.9999909957939888|
  |collected|0.9999907658056393|
  |  kingdom|0.9999896797527402|
  | presumed|0.9999806586578037|
  | patients|0.9999778970248504|
  |    azats|0.9999718388241235|
  |  opening| 0.999969723774294|
  |  holdout|0.9999685636131942|
  | contrast|0.9999677676714386|
  +---------+------------------+
>>> # Find 10 synonyms of a different word
>>> synonyms2 = model.findSynonyms("crime", 10)
>>> synonyms2.show()
  +-----------+------------------+
  |       word|        similarity|
  +-----------+------------------+
  |    peaceful|0.9999983523475047|
  |    democracy|0.9999964568156694|
  |        areas| 0.999994036518118|
  |    miniscule|0.9999920828755365|
  |         lame|0.9999877327660102|
  |      strikes|0.9999877253180771|
  |  terminology|0.999839393584438|
  |        wrath|0.9999829348358952|
  |      divided| 0.999982619125983|
  |      hillary|0.9999795817857984|
```

```
+-----------+------------------+
```

The major difference between the Scala implementation and Python implementation is that in the Python implementation, the stop words have not been removed. That is because that functionality is not available in Python API of the Spark Machine Library. Because of this difference, the list of synonyms generated by Scala program and Python program are different.

References

For more information refer the following links:

- `http://archive.ics.uci.edu/ml/index.html`
- `http://archive.ics.uci.edu/ml/datasets/Wine+Quality`
- `http://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups`

Summary

Spark provides a very powerful core data processing framework and the Spark machine learning library makes use of all the core features of Spark and Spark libraries such as Spark SQL, in addition to its rich set of machine learning algorithms. This chapter covered some of the very common prediction use cases and classification use cases with Scala and Python implementations using the Spark machine learning library with a few lines of code. These wine quality prediction, wine classification, spam filter, and synonym finder machine learning use cases have great potential to be developed into full-blown real-world use cases. Spark 2.0 brings flexibility to model creation, pipeline creation, and their usage in different programs written in a different languages by enabling the model and pipeline persistence.

Pair-wise relationships are very common in real-world use cases. Backed by a strong mathematical theoretical base, computer scientists have developed many data structures and the algorithms that are going with it falling under the subject of Graph Theory. These data structures and algorithms have huge applicability in applications such as social networking websites, scheduling problems, and many other applications. Graph processing is very computationally intensive and distributed data processing paradigms such as Spark are ideal for doing such computations. The Spark GraphX library built on top of Spark is a collection of graph processing APIs. The next chapter is going to take a look at Spark GraphX.

8
Spark Graph Processing

A graph is a mathematical concept and a data structure in computer science. It has huge applications in many real-world use cases. It is used to model a pair-wise relationship between entities. An entity here is known as a vertex and two vertices are connected by an edge. A graph comprises a collection of vertices, and the edges connecting them.

Conceptually, it is a deceptively simple abstraction, but when it comes to processing a huge number of vertices and edges, it is computationally intensive and consumes a lot of processing time and computing resources. Here is a representation of a graph with four vertices and three edges:
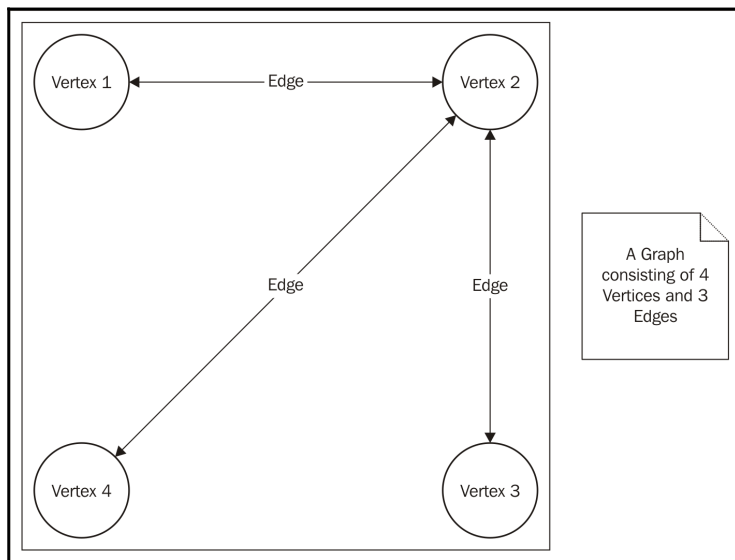
Figure 1

We will cover the following topics in this chapter:

- Graphs and their uses
- The GraphX library
- The PageRank algorithm
- The Connected component algorithm
- GraphFrames
- Graph Queries

Understanding graphs and their usage

There are numerous application constructs that can be modeled as graphs. In a social networking application, the relationship between users can be modeled as a graph in which the users form the vertices of the graph and the relationships between users form the edges of the graph. In a multi-stage job scheduling application, the individual tasks form the vertices of the graph and the sequencing of the tasks forms the edges. In a road traffic modeling system, the towns form the vertices of the graph and the roads connecting the towns form the edges.

The edges of a given graph have a very important property, namely *the direction of the connection*. In many use cases, the direction of the connection doesn't matter. The case of connectivity between cities by roads is one such example. But if the use case is to produce driving directions within a city, the connectivity between traffic junctions has a direction. Take any two traffic junctions and there will be road connectivity, but it is also possible that it is a one-way road. So it all depends on the direction the traffic is flowing. If the road is open to traffic from traffic junction J1 to J2 but closed from J2 to J1, then the graph of driving directions will have a connectivity from J1 to J2 and not from J2 to J1. In such cases, the edge connecting J1 and J2 has a direction. If the road between J2 and J3 is open in both directions, then the edge connecting J2 and J3 has no direction. A graph in which all the edges have a direction is called a **directed graph**.

> When representing a graph pictorially, it is mandatory to give the direction on the edges of the directed graph. If it is not a directed graph, the edge can be represented without any direction at all or with direction to both sides. This is up to the individual's choice. *Figure 1* is not a directed graph, but is represented with directions to both the vertices that the edge is connecting.

In *Figure 2*, the relationship between two users in a social networking application use case is represented as a graph. Users form the vertices and the relationships between the users form the edges. User A follows User B. At the same time, User A is the son of User B. In this graph, there are two parallel edges sharing the same source and destination vertices. A graph containing parallel edges is called a multigraph. The graph shown in *Figure 2* is also a directed graph. This is a good example of a **directed multigraph**.
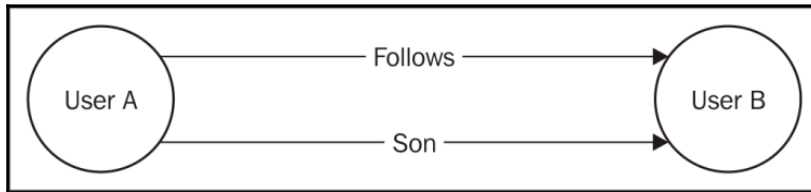
Figure 2

In real-world use cases, the vertices and edges of a graph represent real-world entities. These entities have properties. For example, in the social connectivity graph of users from a social networking application, the users form the vertices and users have many properties such as name, e-mail, phone number, and so on. Similarly, the relationships between the users form the edges of the graph and the edges connecting user vertices can have properties such as relationship. Any graph processing application library should be flexible enough to attach any kind of property to the vertices and edges of a graph.

The Spark GraphX library

For graph processing, many libraries are available in the open source world. Giraph, Pregel, GraphLab, and Spark GraphX are some of them. Spark GraphX is one of the recent entrants into this space.

What is so special about Spark GraphX? Spark GraphX is a graph processing library built on top of the Spark data processing framework. Compared to the other graph processing libraries, Spark GraphX has a real advantage. It can make use of all the data processing capabilities of Spark. However, in reality, the performance of graph processing algorithms is not the only aspect that needs consideration.

In many applications, the data that needs to be modeled as a graph does not exist in that form naturally. In many use cases, more than the graph processing, lots of processor time and other computing resources are expended to get the data in the right format so that the graph processing algorithms can be applied. This is the sweet spot where the combination of the Spark data processing framework and the Spark GraphX library deliver their value. The data processing jobs to make the data ready to be consumed by the Spark GraphX can be easily done using the plethora of tools available in the Spark toolkit. In summary, the Spark GraphX library, which is part of the Spark family, combines the power of the core data processing capabilities of Spark and a very easy-to-use graph processing library.

Revisit the bigger picture once again, as given in *Figure 3*, to set the context and see what is being discussed here before getting into the use cases. Unlike other chapters, in this chapter, the code samples will only be done in Scala because the Spark GraphX library only has a Scala API available at the moment.
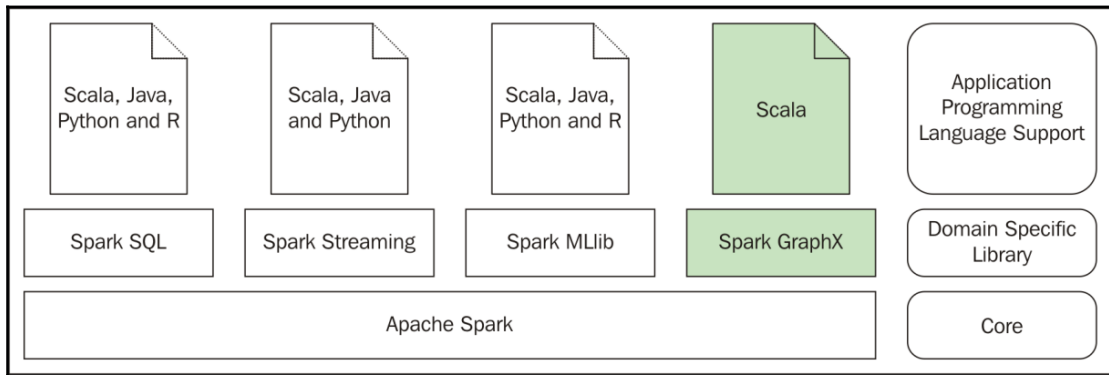
Figure 3

GraphX overview

In any real-world use case, it is easy to understand the concept of a graph comprising vertices and edges. But when it comes to the implementation, this is not a data structure that is very well understood by even good designers and programmers. The reason is simple: unlike other ubiquitous data structures such as list, set, map, queue, and so on, graphs are not commonly used in most applications. Taking this into consideration, the concepts are introduced slowly and steadily, one step at a time, with simple and trivial examples, before taking up some real-world use cases.

The most important aspect of the Spark GraphX library is a data type, Graph, which extends the Spark **resilient distributed dataset** (**RDD**) and introduces a new graph abstraction. The graph abstraction in Spark GraphX is a directed multigraph with properties attached to all the vertices and edges. The properties for each of these vertices and edges can be user defined types that are supported by the Scala type system. These types are parameterized in the Graph type. A given graph may be required to have different data types for vertices or edges. This is possible by using a type system related by an inheritance hierarchy. In addition to all these basic ground rules, the library includes a collection of graph builders and algorithms.

A vertex in a graph is identified by a unique 64-bit long identifier, `org.apache.spark.graphx.VertexId`. Instead of the VertexId type, a simple Scala type, Long, can also be used. In addition to that, vertices can take any type as a property. An edge in a graph should have a source vertex identifier, a destination vertex identifier, and any type as a property.

Figure 4 shows a graph with a vertex property as a String type and an edge property as a String type. In addition to the properties, each vertex has a unique identifier and each edge has a source vertex number and destination vertex number.
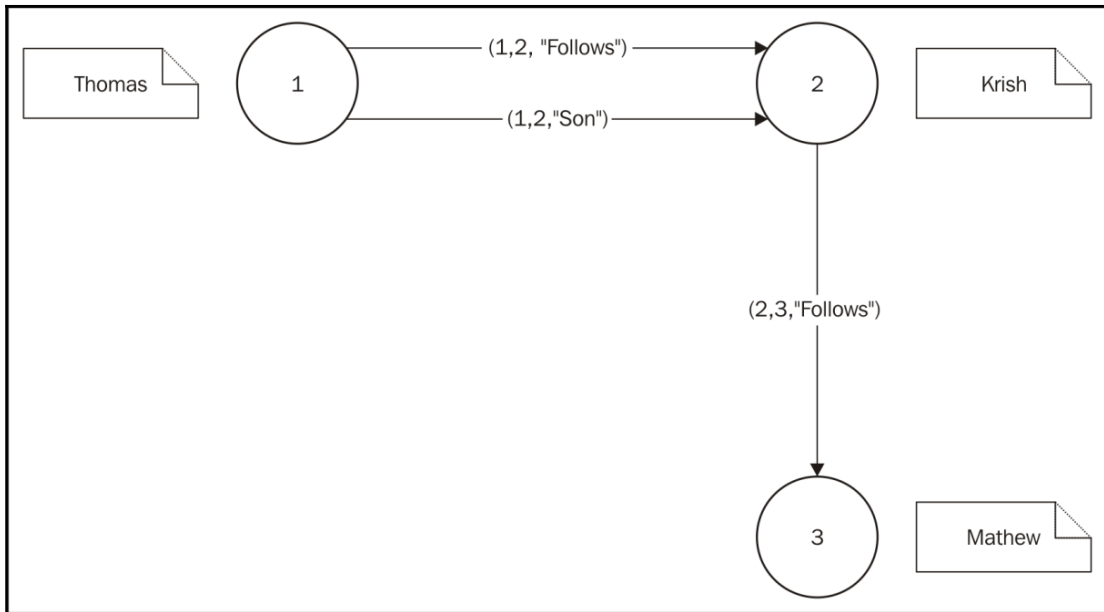
Figure 4

When processing a graph, there are methods to get the vertices and edges. But these independent objects of a graph in isolation may not be sufficient while doing processing.

A vertex has its unique identifier and a property, as stated previously. An edge is uniquely identified by its source and destination vertices. To easily process each edge in graph processing applications, the triplet abstraction of the Spark GraphX library provides an easy way to access the properties of the source vertex, destination vertex, and the edge from a single object.

The following Scala code snippet is used to create the graph shown in *Figure 4* using the Spark GraphX library. After creating the graph, many methods are invoked on the graph that expose various properties of the graph. At the Scala REPL prompt, try the following statements:

```
scala> import org.apache.spark._
   import org.apache.spark._
scala> import org.apache.spark.graphx._
   import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
   import org.apache.spark.rdd.RDD
scala> //Create an RDD of users containing tuple values with a mandatory
Long and another String type as the property of the vertex
scala> val users: RDD[(Long, String)] = sc.parallelize(Array((1L,
"Thomas"), (2L, "Krish"),(3L, "Mathew")))
users: org.apache.spark.rdd.RDD[(Long, String)] = ParallelCollectionRDD[0]
at parallelize at <console>:31
scala> //Created an RDD of Edge type with String type as the property of
the edge
scala> val userRelationships: RDD[Edge[String]] =
sc.parallelize(Array(Edge(1L, 2L, "Follows"),    Edge(1L, 2L,
"Son"),Edge(2L, 3L, "Follows")))
userRelationships:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[1] at parallelize at <console>:31
scala> //Create a graph containing the vertex and edge RDDs as created
before
scala> val userGraph = Graph(users, userRelationships)
userGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@ed5cf29
scala> //Number of edges in the graph
scala> userGraph.numEdges
      res3: Long = 3
scala> //Number of vertices in the graph
scala> userGraph.numVertices
      res4: Long = 3
scala> //Number of edges coming to each of the vertex.
scala> userGraph.inDegrees
```

```
res7: org.apache.spark.graphx.VertexRDD[Int] = VertexRDDImpl[19] at RDD at
VertexRDD.scala:57
scala> //The first element in the tuple is the vertex id and the second
element in the tuple is the number of edges coming to that vertex
scala> userGraph.inDegrees.foreach(println)
      (3,1)
      (2,2)
scala> //Number of edges going out of each of the vertex.
scala> userGraph.outDegrees
res9: org.apache.spark.graphx.VertexRDD[Int] = VertexRDDImpl[23] at RDD at
VertexRDD.scala:57
scala> //The first element in the tuple is the vertex id and the second
element in the tuple is the number of edges going out of that vertex
scala> userGraph.outDegrees.foreach(println)
      (1,2)
      (2,1)
scala> //Total number of edges coming in and going out of each vertex.
scala> userGraph.degrees
res12: org.apache.spark.graphx.VertexRDD[Int] = VertexRDDImpl[27] at RDD at
VertexRDD.scala:57
scala> //The first element in the tuple is the vertex id and the second
element in the tuple is the total number of edges coming in and going out
of that vertex.
scala> userGraph.degrees.foreach(println)
      (1,2)
      (2,3)
      (3,1)
scala> //Get the vertices of the graph
scala> userGraph.vertices
res11: org.apache.spark.graphx.VertexRDD[String] = VertexRDDImpl[11] at RDD
at VertexRDD.scala:57
scala> //Get all the vertices with the vertex number and the property as a
tuple
scala> userGraph.vertices.foreach(println)
      (1,Thomas)
      (3,Mathew)
      (2,Krish)
scala> //Get the edges of the graph
scala> userGraph.edges
res15: org.apache.spark.graphx.EdgeRDD[String] = EdgeRDDImpl[13] at RDD at
EdgeRDD.scala:41
scala> //Get all the edges properties with source and destination vertex
numbers
scala> userGraph.edges.foreach(println)
      Edge(1,2,Follows)
      Edge(1,2,Son)
      Edge(2,3,Follows)
scala> //Get the triplets of the graph
```

```
scala> userGraph.triplets
res18:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.EdgeTriplet[String,String]
] = MapPartitionsRDD[32] at mapPartitions at GraphImpl.scala:48
scala> userGraph.triplets.foreach(println)
    ((1,Thomas),(2,Krish),Follows)
    ((1,Thomas),(2,Krish),Son)
    ((2,Krish),(3,Mathew),Follows)
```

Readers will be familiar with Spark programming using RDDs. The preceding code snippet elucidated the process of constructing the vertices and edges of a graph using RDDs. RDDs can be constructed using data persisted in various data stores. In real-world use cases, most of the time the data will come from external sources, such as NoSQL data stores, and there are ways to construct RDDs using such data. Once the RDDs are constructed, graphs can be constructed using that.

The preceding code snippet also explained the various methods available with the graph to get all the required details of a given graph. The teaser use case covered here is a very small graph in terms of size. In real-world use cases, the number of vertices and edges of a graph can be in the millions. Since all these abstractions are implemented as RDDs, all the inherent goodness of immutability, partitioning, distribution, and parallel processing comes out of the box, hence making graph processing highly scalable. Finally, the following tables show how the vertices and edges are represented:

Vertex table:

VertexId	Vertex property
1	Thomas
2	Krish
3	Mathew

Edge table:

Source VertexId	Destination VertexId	Edge property
1	2	Follows
1	2	Son
2	3	Follows

Triplet table:

Source VertexId	Destination VertexId	Source vertex Property	Edge property	Destination vertex property
1	2	Thomas	Follows	Krish
1	2	Thomas	Son	Krish
2	3	Krish	Follows	Mathew

It is important to note that these tables are only for explanation purposes. The real internal representation follows the rules and regulations of RDD representation.

If anything is represented as an RDD, it is bound to get partitioned and distributed. But if the partitioning and distribution are done freely, without any control for the graph, then it is going to be suboptimal when it comes to graph processing performance. Because of that, the creators of the Spark GraphX library have thought through this problem well in advance and implemented a graph partitioning strategy in order to have an optimized representation of the graph as RDDs.

Graph partitioning

It is important to understand a little bit about how the graph RDDs are partitioned and distributed across various partitions. This will be useful for advanced optimizations that determine the partition and distribution of the various RDDs that are the constituent parts of a graph.

In general, there are three RDDs for a given graph. Apart from the vertex RDD and the edge RDD, one more RDD is used internally, and that is the routing RDD. To have optimal performance, all the vertices needed to form a given edge are kept in the same partition where the edge is stored. If a given vertex is participating in multiple edges and these edges are located in different partitions, then this particular vertex can be stored in multiple partitions.

To keep track of the partitions where a given vertex is stored redundantly, a routing RDD is also maintained, containing the vertex details and the partitions in which each vertex is available.

Figure 5 explains this:
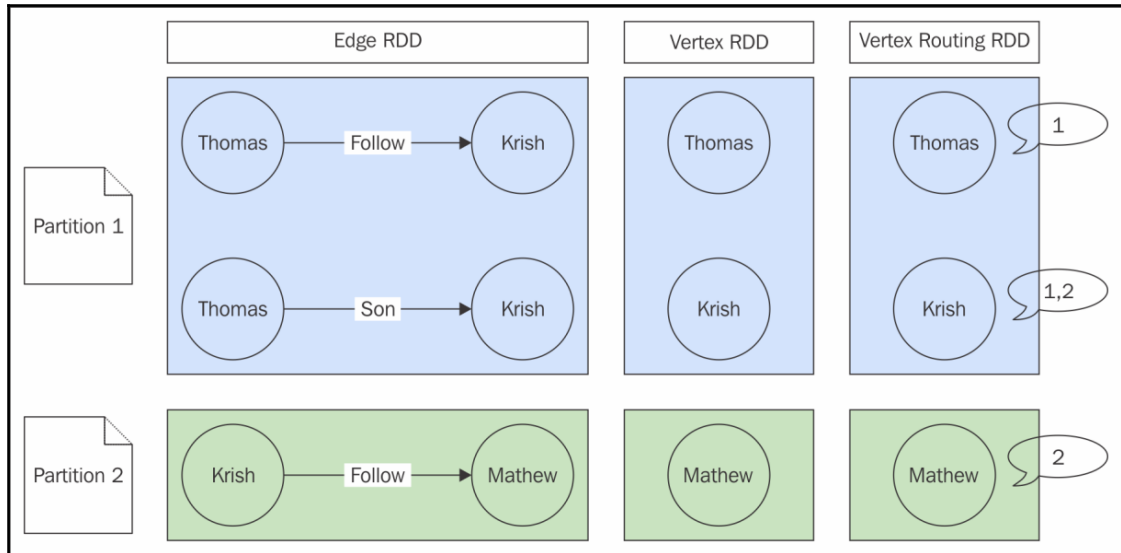
Figure 5

In *Figure 5*, assume that the edges are partitioned into partitions 1 and 2. Also assume that the vertices are partitioned into partitions 1 and 2.

In partition 1, all the vertices required for the edges are available locally. But in partition 2, only one vertex for the edge is available locally. So the missing vertex is also stored in partition 2 so that all the required vertices are available locally.

To keep track of the replications, the vertex routing RDD maintains the partition numbers where a given vertex is available. In *Figure 5*, in the vertex routing RDD, callout symbols are used to show the partitions in which these vertices are replicated. In this way, while processing the edges or triplets, all the information related to the constituent vertices is available locally and performance will be highly optimal. Since the RDDs are immutable, the problems associated with information getting changed are removed, even if they are stored in multiple partitions.

Graph processing

The constituent elements of a graph exposed to the users are the vertex RDD and the edge RDD. Just like any other data structure, a graph also undergoes lots of changes because of the change in the underlying data. To make the required graph operations to support various use cases, there are many algorithms available, using which the data hidden in the graph data structure can be processed to produce the desired business outcomes. Before getting into the algorithms to process a graph, it is good to understand some of the basics of graph processing using an air travel use case.

Assume that a person is trying to find a cheap return air ticket from Manchester to Bangalore. In the travel preferences, this person has mentioned that he/she doesn't care about the number of stops but the price should be the lowest. Assume that the air ticket reservation system has picked up the same stops for both the onward and the return journey and produced the following routes or journey legs with the cheapest price:

Manchester → London → Colombo → Bangalore

Bangalore → Colombo → London → Manchester

This route plan is a perfect example of a graph. If the onward journey is considered as one graph and the return journey is considered as another graph, the return journey graph can be produced by reversing the onward journey graph. At the Scala REPL prompt, try the following statements:

```
scala> import org.apache.spark._
import org.apache.spark._
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD
scala> //Create the vertices with the stops
scala> val stops: RDD[(Long, String)] = sc.parallelize(Array((1L,
"Manchester"), (2L, "London"),(3L, "Colombo"), (4L, "Bangalore")))
stops: org.apache.spark.rdd.RDD[(Long, String)] = ParallelCollectionRDD[33]
at parallelize at <console>:38
scala> //Create the edges with travel legs
scala> val legs: RDD[Edge[String]] = sc.parallelize(Array(Edge(1L, 2L,
"air"),    Edge(2L, 3L, "air"),Edge(3L, 4L, "air")))
legs: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[34] at parallelize at <console>:38
scala> //Create the onward journey graph
scala> val onwardJourney = Graph(stops, legs)onwardJourney:
org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@190ec769scala>
```

```
onwardJourney.triplets.map(triplet => (triplet.srcId, (triplet.srcAttr,
triplet.dstAttr))).sortByKey().collect().foreach(println)
(1,(Manchester,London))
(2,(London,Colombo))
(3,(Colombo,Bangalore))
scala> val returnJourney = onwardJourney.reversereturnJourney:
org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@60035f1e
scala> returnJourney.triplets.map(triplet => (triplet.srcId,
(triplet.srcAttr,triplet.dstAttr))).sortByKey(ascending=false).collect().fo
reach(println)
(4,(Bangalore,Colombo))
(3,(Colombo,London))
(2,(London,Manchester))
```

The source and destination of the onward journey legs are reversed in the return journey legs. When a graph is reversed, only the source and destination vertices of the edges are reversed and the identity of the vertices remains the same.

In other words, the vertex identifiers of each of the vertices remain the same. While processing a graph, it is important to know the names of the triplet attributes. They are useful for writing programs and processing the graph. As a continuation of the same Scala REPL session, try the following statements:

```
scala> returnJourney.triplets.map(triplet =>
(triplet.srcId,triplet.dstId,triplet.attr,triplet.srcAttr,triplet.dstAttr))
.foreach(println)
(2,1,air,London,Manchester)
(3,2,air,Colombo,London)
(4,3,air,Bangalore,Colombo)
```

The following table gives the list of attributes of a triplet that can be used to process a graph and extract the required data from the graph. The preceding code snippet and the following table may be cross-verified to fully understand:

Triplet attribute	Description
srcId	Source vertex identifier
dstId	Destination vertex identifier
attr	Edge property
srcAttr	Source vertex property
dstAttr	Destination vertex property

In a graph, vertices are RDDs and edges are RDDs, and just by virtue of that, transformations are possible.

Now, to demonstrate graph transformations, the same use case is used, with a slight twist. Assume that a travel agent is getting special discount prices from the airline companies for selected routes. The travel agent decides to keep the discount and offer the market price to his/her customers, and for this purpose he/she adds 10% to the price given by the airline company. This travel agent has noticed that the airport names are being displayed inconsistently and wanted to make sure that there is consistent representation when displayed throughout the website and decides to change all the stop names to upper case. As a continuation of the same Scala REPL session, try the following statements:

```scala
scala> // Create the vertices
scala> val stops: RDD[(Long, String)] = sc.parallelize(Array((1L,
"Manchester"), (2L, "London"),(3L, "Colombo"), (4L, "Bangalore")))
stops: org.apache.spark.rdd.RDD[(Long, String)] = ParallelCollectionRDD[66]
at parallelize at <console>:38
scala> //Create the edges
scala> val legs: RDD[Edge[Long]] = sc.parallelize(Array(Edge(1L, 2L, 50L),
Edge(2L, 3L, 100L),Edge(3L, 4L, 80L)))
legs: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Long]] =
ParallelCollectionRDD[67] at parallelize at <console>:38
scala> //Create the graph using the vertices and edges
scala> val journey = Graph(stops, legs)
journey: org.apache.spark.graphx.Graph[String,Long] =
org.apache.spark.graphx.impl.GraphImpl@8746ad5
scala> //Convert the stop names to upper case
scala> val newStops = journey.vertices.map {case (id, name) => (id,
name.toUpperCase)}
newStops: org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
String)] = MapPartitionsRDD[80] at map at <console>:44
scala> //Get the edges from the selected journey and add 10% price to the
original price
scala> val newLegs = journey.edges.map { case Edge(src, dst, prop) =>
Edge(src, dst, (prop + (0.1*prop))) }
newLegs: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Double]] =
MapPartitionsRDD[81] at map at <console>:44
scala> //Create a new graph with the original vertices and the new edges
scala> val newJourney = Graph(newStops, newLegs)
newJourney: org.apache.spark.graphx.Graph[String,Double] =
org.apache.spark.graphx.impl.GraphImpl@3c929623
scala> //Print the contents of the original graph
scala> journey.triplets.foreach(println)
((1,Manchester),(2,London),50)
((3,Colombo),(4,Bangalore),80)
((2,London),(3,Colombo),100)
scala> //Print the contents of the transformed graph
```

```
scala>  newJourney.triplets.foreach(println)
((2,LONDON),(3,COLOMBO),110.0)
((3,COLOMBO),(4,BANGALORE),88.0)
((1,MANCHESTER),(2,LONDON),55.0)
```

In essence, these transformations are truly RDD transformations. If there is a conceptual understanding of how these different RDDs are cobbled together to form a graph, any programmer with RDD programming proficiency will be able to do graph processing very well. This is another testament to the power of the unified programming model of Spark.

The preceding use case did the map transformation on vertex and edge RDDs. Similarly, filter transformations are another useful type that is commonly used. Apart from these, all the transformations and actions can be used to process the vertex and edge RDDs.

Graph structure processing

In the previous section, one type of graph processing is done by individually processing the required vertices or edges. One disadvantage of this approach is that the processing is going through three different stages, as follows:

- Extract vertices or edges from the graph
- Process the vertices or edges
- Re-create a new graph with the processed vertices and edges

This is tedious and prone to user programming errors. To circumvent this problem, there are some structural operators available in the Spark GraphX library that let users process the graph as an individual unit that produces a new graph.

One important structural operation has already been discussed in the previous section, which is the reversal of a graph producing a new graph with all the directions of the edges reversed. Another frequently used structural operation is the extraction of a subgraph from a given graph. The resultant subgraph can be the entire parent graph itself or a subset of the parent graph, depending on the operation that is done on the parent graph.

When creating a graph from data from external sources, there is a possibility that the edges may have invalid vertices. This is very much a possibility if the vertices and the edges are created from the data coming from two different sources or different applications. With these vertices and edges, if a graph is created, some of the edges will have invalid vertices, and processing will result in unexpected outcomes. The following is a use case where some of the edges containing invalid vertices and pruning are done to get rid of that using a structural operator. At the Scala REPL prompt, try the following statements:

```
scala> import org.apache.spark._
  import org.apache.spark._
scala> import org.apache.spark.graphx._
  import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
  import org.apache.spark.rdd.RDD
scala> //Create an RDD of users containing tuple values with a mandatory
Long and another String type as the property of the vertex
scala> val users: RDD[(Long, String)] = sc.parallelize(Array((1L,
"Thomas"), (2L, "Krish"),(3L, "Mathew")))
users: org.apache.spark.rdd.RDD[(Long, String)] =
ParallelCollectionRDD[104] at parallelize at <console>:45
scala> //Created an RDD of Edge type with String type as the property of
the edge
scala> val userRelationships: RDD[Edge[String]] =
sc.parallelize(Array(Edge(1L, 2L, "Follows"), Edge(1L, 2L, "Son"),Edge(2L,
3L, "Follows"), Edge(1L, 4L, "Follows"), Edge(3L, 4L, "Follows")))
userRelationships:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[105] at parallelize at <console>:45
scala> //Create a vertex property object to fill in if an invalid vertex id
is given in the edge
scala> val missingUser = "Missing"
missingUser: String = Missing
scala> //Create a graph containing the vertex and edge RDDs as created
before
scala> val userGraph = Graph(users, userRelationships, missingUser)
userGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@43baf0b9
scala> //List the graph triplets and find some of the invalid vertex ids
given and for them the missing vertex property is assigned with the value
"Missing"
scala> userGraph.triplets.foreach(println)
      ((3,Mathew),(4,Missing),Follows)
      ((1,Thomas),(2,Krish),Son)
      ((2,Krish),(3,Mathew),Follows)
      ((1,Thomas),(2,Krish),Follows)
      ((1,Thomas),(4,Missing),Follows)
scala> //Since the edges with the invalid vertices are invalid too, filter
out those vertices and create a valid graph. The vertex predicate here can
be any valid filter condition of a vertex. Similar to vertex predicate, if
the filtering is to be done on the edges, instead of the vpred, use epred
as the edge predicate.
scala> val fixedUserGraph = userGraph.subgraph(vpred = (vertexId,
attribute) => attribute != "Missing")
fixedUserGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@233b5c71
```

```
scala> fixedUserGraph.triplets.foreach(println)
    ((2,Krish),(3,Mathew),Follows)
    ((1,Thomas),(2,Krish),Follows)
    ((1,Thomas),(2,Krish),Son)
```

In huge graphs, at times depending on the use case, there can be a whole lot of parallel edges. In some use cases, it is possible to combine the data of the parallel edges and maintain only one edge instead of maintaining lots of parallel edges. In the preceding use case, the final graph without any invalid edges, there are parallel edges, one with the property `Follows` and the other with `Son`, which have the same source and destination vertices.

It is fine to combine these parallel edges into one single edge with the property concatenated from the parallel edges, which will reduce the number of edges without losing information. This is accomplished by the groupEdges structural operation of the graph. As a continuation of the same Scala REPL session, try the following statements:

```
scala> // Import the partition strategy classes
scala> import org.apache.spark.graphx.PartitionStrategy._
import org.apache.spark.graphx.PartitionStrategy._
scala> // Partition the user graph. This is required to group the edges
scala> val partitionedUserGraph =
fixedUserGraph.partitionBy(CanonicalRandomVertexCut)
partitionedUserGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@5749147e
scala> // Generate the graph without parallel edges and combine the
properties of duplicate edges
scala> val graphWithoutParallelEdges = partitionedUserGraph.groupEdges((e1,
e2) => e1 + " and " + e2)
graphWithoutParallelEdges: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@16a4961f
scala> // Print the details
scala> graphWithoutParallelEdges.triplets.foreach(println)
((1,Thomas),(2,Krish),Follows and Son)
((2,Krish),(3,Mathew),Follows)
```

The preceding structural change in the graph reduced the number of edges by grouping the edges. When the edge property is numerical, and if it makes sense to consolidate by aggregating them, then also reduce the number of edges by removing the parallel edges, which can reduce the graph processing time considerably.

> One important point to note in this code snippet is that the graph has been partitioned before the group-by operation on the edges.

By default, the edges and the constituent vertices of a given graph need not be co-located in the same partition. For the group-by operation to work, all the parallel edges have to be located on the same partition. The CanonicalRandomVertexCut partition strategy makes sure that colocation happens for all the edges between two vertices, irrespective of direction.

There are some more structural operators available in the Spark GraphX library and a consultation of the Spark documentation will give a good insight into them. They can be used depending on the use case.

Tennis tournament analysis

Since the basic graph processing fundamentals are in place, now it is time to take up a real-world use case that uses graphs. Here, a tennis tournament's results are modeled using a graph. The Barclays ATP World Tour 2015 singles competition results are modeled using a graph. The vertices contain the player details and the edges contain the individual matches played. The edges are formed in such a way that the source vertex is the player who won the match and the destination vertex is the player who lost the match. The edge property contains the type of the match, the points the winner got in the match, and the head-to-head count of the players in the match. The points system used here is fictitious and is nothing but a weight earned by the winner in that particular match. The initial group matches carried the least weight, the semi-final matches carried more weight, and the final match carried the most weight. With this way of modeling the results, find out the following details by processing the graph:

- List all the match details.
- List all the matches with player names, the match type, and the result.
- List all the Group 1 winners with the points in the match.
- List all the Group 2 winners with the points in the match.
- List all the semi-final winners with the points in the match.
- List the final winner with the points in the match.
- List the players with the total points they earned in the whole tournament.
- List the winner of the match by finding the highest number of points scored by the player.
- In the group-based matches, because of the round robin scheme of draws, it is possible that the same players can meet more than once. Find if there are any such players who have played each other more than once in this tournament.
- List the players who have won at least one match.
- List the players who have lost at least one match.

- List the players who have won at least one match and lost at least one match.
- List the players who have no wins at all.
- List the players who have no losses at all.

Those who are not familiar with the game of tennis have no need to worry because the rules of the games are not discussed here and are not required to understand this use case. For all practical purposes, it is to be taken only as a game played between two people, where one wins and the other loses. At the Scala REPL prompt, try the following statements:

```
scala> import org.apache.spark._
  import org.apache.spark._
scala> import org.apache.spark.graphx._
  import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
  import org.apache.spark.rdd.RDD
scala> //Define a property class that is going to hold all the properties
of the vertex which is nothing but player information
scala> case class Player(name: String, country: String)
      defined class Player
scala> // Create the player vertices
scala> val players: RDD[(Long, Player)] = sc.parallelize(Array((1L,
Player("Novak Djokovic", "SRB")), (3L, Player("Roger Federer", "SUI")),(5L,
Player("Tomas Berdych", "CZE")), (7L, Player("Kei Nishikori", "JPN")),
(11L, Player("Andy Murray", "GBR")),(15L, Player("Stan Wawrinka",
"SUI")),(17L, Player("Rafael Nadal", "ESP")),(19L, Player("David Ferrer",
"ESP"))))
players: org.apache.spark.rdd.RDD[(Long, Player)] =
ParallelCollectionRDD[145] at parallelize at <console>:57
scala> //Define a property class that is going to hold all the properties
of the edge which is nothing but match information
scala> case class Match(matchType: String, points: Int, head2HeadCount:
Int)
      defined class Match
scala> // Create the match edges
scala> val matches: RDD[Edge[Match]] = sc.parallelize(Array(Edge(1L, 5L,
Match("G1", 1,1)), Edge(1L, 7L, Match("G1", 1,1)), Edge(3L, 1L, Match("G1",
1,1)), Edge(3L, 5L, Match("G1", 1,1)), Edge(3L, 7L, Match("G1", 1,1)),
Edge(7L, 5L, Match("G1", 1,1)), Edge(11L, 19L, Match("G2", 1,1)), Edge(15L,
11L, Match("G2", 1, 1)), Edge(15L, 19L, Match("G2", 1, 1)), Edge(17L, 11L,
Match("G2", 1, 1)), Edge(17L, 15L, Match("G2", 1, 1)), Edge(17L, 19L,
Match("G2", 1, 1)), Edge(3L, 15L, Match("S", 5, 1)), Edge(1L, 17L,
Match("S", 5, 1)), Edge(1L, 3L, Match("F", 11, 1))))
matches: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Match]] =
ParallelCollectionRDD[146] at parallelize at <console>:57
scala> //Create a graph with the vertices and edges
scala> val playGraph = Graph(players, matches)
playGraph: org.apache.spark.graphx.Graph[Player,Match] =
```

```
org.apache.spark.graphx.impl.GraphImpl@30d4d6fb
```

The graph containing the tennis tournament has been created, and from now on, all that is going to be done is the processing of this base graph and extracting information from it to fulfill the requirements of the use cases:

```
scala> //Print the match details
scala> playGraph.triplets.foreach(println)
((15,Player(Stan Wawrinka,SUI)),(11,Player(Andy Murray,GBR)),Match(G2,1,1))
((15,Player(Stan Wawrinka,SUI)),(19,Player(David
Ferrer,ESP)),Match(G2,1,1))
((7,Player(Kei Nishikori,JPN)),(5,Player(Tomas Berdych,CZE)),Match(G1,1,1))
((1,Player(Novak Djokovic,SRB)),(7,Player(Kei
Nishikori,JPN)),Match(G1,1,1))
((3,Player(Roger Federer,SUI)),(1,Player(Novak
Djokovic,SRB)),Match(G1,1,1))
((1,Player(Novak Djokovic,SRB)),(3,Player(Roger
Federer,SUI)),Match(F,11,1))
((1,Player(Novak Djokovic,SRB)),(17,Player(Rafael Nadal,ESP)),Match(S,5,1))
((3,Player(Roger Federer,SUI)),(5,Player(Tomas Berdych,CZE)),Match(G1,1,1))
((17,Player(Rafael Nadal,ESP)),(11,Player(Andy Murray,GBR)),Match(G2,1,1))
((3,Player(Roger Federer,SUI)),(7,Player(Kei Nishikori,JPN)),Match(G1,1,1))
((1,Player(Novak Djokovic,SRB)),(5,Player(Tomas
Berdych,CZE)),Match(G1,1,1))
((17,Player(Rafael Nadal,ESP)),(15,Player(Stan
Wawrinka,SUI)),Match(G2,1,1))
((11,Player(Andy Murray,GBR)),(19,Player(David Ferrer,ESP)),Match(G2,1,1))
((3,Player(Roger Federer,SUI)),(15,Player(Stan Wawrinka,SUI)),Match(S,5,1))
((17,Player(Rafael Nadal,ESP)),(19,Player(David Ferrer,ESP)),Match(G2,1,1))
scala> //Print matches with player names and the match type and the result
scala> playGraph.triplets.map(triplet => triplet.srcAttr.name + " won over
" + triplet.dstAttr.name + " in  " + triplet.attr.matchType + "
match").foreach(println)
        Roger Federer won over Tomas Berdych in  G1 match
        Roger Federer won over Kei Nishikori in  G1 match
        Novak Djokovic won over Roger Federer in  F match
        Novak Djokovic won over Rafael Nadal in  S match
        Roger Federer won over Stan Wawrinka in  S match
        Rafael Nadal won over David Ferrer in  G2 match
        Kei Nishikori won over Tomas Berdych in  G1 match
        Andy Murray won over David Ferrer in  G2 match
        Stan Wawrinka won over Andy Murray in  G2 match
        Stan Wawrinka won over David Ferrer in  G2 match
        Novak Djokovic won over Kei Nishikori in  G1 match
        Roger Federer won over Novak Djokovic in  G1 match
        Rafael Nadal won over Andy Murray in  G2 match
        Rafael Nadal won over Stan Wawrinka in  G2 match
        Novak Djokovic won over Tomas Berdych in  G1 match
```

It is worth noticing here that the usage of triplets in graphs comes in handy for extracting all the required data elements of a given tennis match, including who was playing, who won, and the match type, from a single object. The following implementations of analysis use cases involve filtering the tennis match records of the tournament. Here, only simple filtering logic is used, but in real-world use cases, any complex logic can be implemented in functions, and that can be passed as arguments to the filter transformations:

```scala
scala> //Group 1 winners with their group total points
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"G1").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).foreach(println)
      (Kei Nishikori,1)
      (Roger Federer,1)
      (Roger Federer,1)
      (Novak Djokovic,1)
      (Novak Djokovic,1)
      (Roger Federer,1)
scala> //Find the group total of the players
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"G1").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).reduceByKey(_+_).foreach(println)
      (Roger Federer,3)
      (Novak Djokovic,2)
      (Kei Nishikori,1)
scala> //Group 2 winners with their group total points
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"G2").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).foreach(println)
      (Rafael Nadal,1)
      (Rafael Nadal,1)
      (Andy Murray,1)
      (Stan Wawrinka,1)
      (Stan Wawrinka,1)
      (Rafael Nadal,1)
```

The following implementations of analysis use cases involve grouping by key and doing summary calculations. It is not limited to just finding the sum of the tennis match record points, as shown in the following use case implementations; rather, user-defined functions can be used to do the calculations as well:

```scala
scala> //Find the group total of the players
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"G2").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).reduceByKey(_+_).foreach(println)
      (Stan Wawrinka,2)
      (Andy Murray,1)
      (Rafael Nadal,3)
```

```
scala> //Semi final winners with their group total points
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"S").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).foreach(println)
      (Novak Djokovic,5)
      (Roger Federer,5)
scala> //Find the group total of the players
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"S").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).reduceByKey(_+_).foreach(println)
      (Novak Djokovic,5)
      (Roger Federer,5)
scala> //Final winner with the group total points
scala> playGraph.triplets.filter(triplet => triplet.attr.matchType ==
"F").map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).foreach(println)
      (Novak Djokovic,11)
scala> //Tournament total point standing
scala> playGraph.triplets.map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).reduceByKey(_+_).foreach(println)
      (Stan Wawrinka,2)

      (Rafael Nadal,3)
      (Kei Nishikori,1)
      (Andy Murray,1)
      (Roger Federer,8)
      (Novak Djokovic,18)
scala> //Find the winner of the tournament by finding the top scorer of the
tournament
scala> playGraph.triplets.map(triplet => (triplet.srcAttr.name,
triplet.attr.points)).reduceByKey(_+_).map{ case (k,v) =>
(v,k)}.sortByKey(ascending=false).take(1).map{ case (k,v) =>
(v,k)}.foreach(println)
      (Novak Djokovic,18)
scala> //Find how many head to head matches held for a given set of players
in the descending order of head2head count
scala> playGraph.triplets.map(triplet => (Set(triplet.srcAttr.name ,
triplet.dstAttr.name) ,
triplet.attr.head2HeadCount)).reduceByKey(_+_).map{case (k,v) =>
(k.mkString(" and "), v)}.map{ case (k,v) => (v,k)}.sortByKey().map{ case
(k,v) => v + " played " + k + " time(s)"}.foreach(println)
      Roger Federer and Novak Djokovic played 2 time(s)
      Roger Federer and Tomas Berdych played 1 time(s)
      Kei Nishikori and Tomas Berdych played 1 time(s)
      Novak Djokovic and Tomas Berdych played 1 time(s)
      Rafael Nadal and Andy Murray played 1 time(s)
      Rafael Nadal and Stan Wawrinka played 1 time(s)
      Andy Murray and David Ferrer played 1 time(s)
```

```
        Rafael Nadal and David Ferrer played 1 time(s)
        Stan Wawrinka and David Ferrer played 1 time(s)
        Stan Wawrinka and Andy Murray played 1 time(s)
        Roger Federer and Stan Wawrinka played 1 time(s)
        Roger Federer and Kei Nishikori played 1 time(s)
        Novak Djokovic and Kei Nishikori played 1 time(s)
        Novak Djokovic and Rafael Nadal played 1 time(s)
```

The following implementations of analysis use cases involve finding unique records from the query. The Spark distinct transformation does that:

```
scala> //List of players who have won at least one match
scala> val winners = playGraph.triplets.map(triplet =>
triplet.srcAttr.name).distinct
winners: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[201] at
distinct at <console>:65
scala> winners.foreach(println)
        Kei Nishikori
        Stan Wawrinka
        Andy Murray
        Roger Federer
        Rafael Nadal
        Novak Djokovic
scala> //List of players who have lost at least one match
scala> val loosers = playGraph.triplets.map(triplet =>
triplet.dstAttr.name).distinct
loosers: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[205] at
distinct at <console>:65
scala> loosers.foreach(println)
        Novak Djokovic
        Kei Nishikori
        David Ferrer
        Stan Wawrinka
        Andy Murray
        Roger Federer
        Rafael Nadal
        Tomas Berdych
scala> //List of players who have won at least one match and lost at least
one match
scala> val wonAndLost = winners.intersection(loosers)
wonAndLost: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[211] at
intersection at <console>:69
scala> wonAndLost.foreach(println)
        Novak Djokovic
        Rafael Nadal
        Andy Murray
        Roger Federer
        Kei Nishikori
```

```
        Stan Wawrinka
scala> //List of players who have no wins at all
scala> val lostAndNoWins = loosers.collect().toSet --
wonAndLost.collect().toSet
lostAndNoWins: scala.collection.immutable.Set[String] = Set(David Ferrer,
Tomas Berdych)
scala> lostAndNoWins.foreach(println)
        David Ferrer
        Tomas Berdych
scala> //List of players who have no loss at all
scala> val wonAndNoLosses = winners.collect().toSet --
loosers.collect().toSet
        wonAndNoLosses: scala.collection.immutable.Set[String] = Set()
scala> //The val wonAndNoLosses returned an empty set which means that
there is no single player in this tournament who have only wins
scala> wonAndNoLosses.foreach(println)
```

In this use case, not much effort has been made to make the results pretty because they are reduced to simple RDD-based structures that can be manipulated however required using the RDD programming techniques that were already covered in the initial chapters of the book.

The highly succinct and uniform programming model of Spark, in conjunction with the Spark GraphX library, helps developers build real-world use cases with very few lines of code. This also demonstrates that once the right graph structure is built with the relevant data, with the supported graph operations, lots of truth that is hidden in the underlying data can be brought to light.

Applying the PageRank algorithm

A research paper, titled *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, by Sergey Brin and Lawrence Page, revolutionized web searching, and Google based its search engine on this concept of PageRank and came to dominate other web search engines.

When searching the web using Google, pages that are ranked highly by its algorithm are displayed. In the context of graphs, instead of web pages, if vertices are ranked based on the same algorithm, lots of new inferences can be made. From the outside, it may sound like this PageRank algorithm is useful only for web searches. But it has immense potential to be applied to many other areas.

In graph parlance, if there is an edge, E, connecting two vertices, from V1 to V2, according to the PageRank algorithm, V2 is more important than V1. In a huge graph of vertices and edges, it is possible to calculate the PageRank of each and every vertex.

The PageRank algorithm can be applied very well to the tennis tournament analysis use case covered in the preceding section. In the graph representation that is adopted here, each match is represented as an edge. The source vertex has the winner's details and the destination vertex has the loser's details. In the game of tennis, if this can be termed as some fictitious importance ranking, then in a given match the winner has higher importance ranking than the loser.

If the graph in the previous use case is taken to demonstrate the PageRank algorithm, then that graph has to be reversed so that the winner of each match becomes the destination vertex of each and every edge. At the Scala REPL prompt, try the following statements:

```scala
scala> import org.apache.spark._
  import org.apache.spark._
scala> import org.apache.spark.graphx._
  import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
  import org.apache.spark.rdd.RDD
scala> //Define a property class that is going to hold all the properties
of the vertex which is nothing but player information
scala> case class Player(name: String, country: String)
      defined class Player
scala> // Create the player vertices
scala> val players: RDD[(Long, Player)] = sc.parallelize(Array((1L,
Player("Novak Djokovic", "SRB")), (3L, Player("Roger Federer", "SUI")),(5L,
Player("Tomas Berdych", "CZE")), (7L, Player("Kei Nishikori", "JPN")),
(11L, Player("Andy Murray", "GBR")),(15L, Player("Stan Wawrinka",
"SUI")),(17L, Player("Rafael Nadal", "ESP")),(19L, Player("David Ferrer",
"ESP"))))
players: org.apache.spark.rdd.RDD[(Long, Player)] =
ParallelCollectionRDD[212] at parallelize at <console>:64
scala> //Define a property class that is going to hold all the properties
of the edge which is nothing but match information
scala> case class Match(matchType: String, points: Int, head2HeadCount:
Int)
      defined class Match
scala> // Create the match edges
scala> val matches: RDD[Edge[Match]] = sc.parallelize(Array(Edge(1L, 5L,
Match("G1", 1,1)), Edge(1L, 7L, Match("G1", 1,1)), Edge(3L, 1L, Match("G1",
1,1)), Edge(3L, 5L, Match("G1", 1,1)), Edge(3L, 7L, Match("G1", 1,1)),
Edge(7L, 5L, Match("G1", 1,1)), Edge(11L, 19L, Match("G2", 1,1)), Edge(15L,
11L, Match("G2", 1, 1)), Edge(15L, 19L, Match("G2", 1, 1)), Edge(17L, 11L,
Match("G2", 1, 1)), Edge(17L, 15L, Match("G2", 1, 1)), Edge(17L, 19L,
Match("G2", 1, 1)), Edge(3L, 15L, Match("S", 5, 1)), Edge(1L, 17L,
Match("S", 5, 1)), Edge(1L, 3L, Match("F", 11, 1))))
matches: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Match]] =
ParallelCollectionRDD[213] at parallelize at <console>:64
scala> //Create a graph with the vertices and edges
```

```
scala> val playGraph = Graph(players, matches)
playGraph: org.apache.spark.graphx.Graph[Player,Match] =
org.apache.spark.graphx.impl.GraphImpl@263cd0e2
scala> //Reverse this graph to have the winning player coming in the
destination vertex
scala> val rankGraph = playGraph.reverse
rankGraph: org.apache.spark.graphx.Graph[Player,Match] =
org.apache.spark.graphx.impl.GraphImpl@7bb131fb
scala> //Run the PageRank algorithm to calculate the rank of each vertex
scala> val rankedVertices = rankGraph.pageRank(0.0001).vertices
rankedVertices: org.apache.spark.graphx.VertexRDD[Double] =
VertexRDDImpl[1184] at RDD at VertexRDD.scala:57
scala> //Extract the vertices sorted by the rank
scala> val rankedPlayers = rankedVertices.join(players).map{case
(id,(importanceRank,Player(name,country))) => (importanceRank,
name)}.sortByKey(ascending=false)
rankedPlayers: org.apache.spark.rdd.RDD[(Double, String)] =
ShuffledRDD[1193] at sortByKey at <console>:76
scala> rankedPlayers.collect().foreach(println)
      (3.382662570589846,Novak Djokovic)
      (3.266079758089846,Roger Federer)
      (0.3908953124999999,Rafael Nadal)
      (0.27431249999999996,Stan Wawrinka)
      (0.1925,Andy Murray)
      (0.1925,Kei Nishikori)
      (0.15,David Ferrer)
      (0.15,Tomas Berdych)
```

If the preceding code is scrutinized carefully, it can be seen that the highest ranked players have won the highest number of matches.

Connected component algorithm

In a graph, finding a subgraph consisting of connected vertices is a very common requirement with tremendous applications. In any graph, two vertices are that connected to each other by paths consisting of one or more edges, and are not connected to any other vertex in the same graph, are called a connected component. For example, in a graph, G, vertex V1 is connected to V2 by an edge and V2 is connected to V3 by another edge. In the same graph, G, vertex V4 is connected to V5 by another edge. In this case V1 and V3 are connected, V4 and V5 are connected and V1 and V5 are not connected. In graph G, there are two connected components. The Spark GraphX library has an implementation of the connected components algorithm.

In a social networking application, if the connections between the users are modeled as a

graph, finding whether a given user is connected to another user is achieved by checking whether there is a connected component with these two vertices. In computer games, maze traversing from point A to point B can be done using a connected components algorithm by modeling the maze junctions as vertices and the paths connecting the junctions as edges in a graph.

In computer networks, checking whether packets can be sent from one IP address to another IP address is achieved by using a connected components algorithm. In logistics applications, such as a courier service, checking whether a packet can be sent from point A to point B is achieved by using a connected components algorithm. *Figure 6* shows a graph with three connected components:
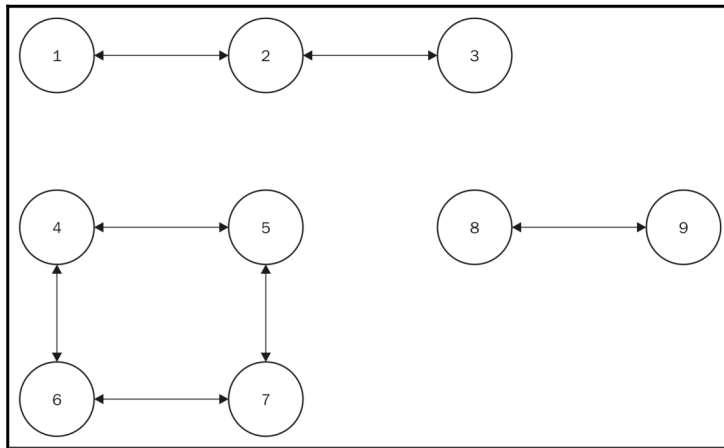
Figure 6

Figure 6 is the pictorial representation of a graph. In it, there are three *clusters* of vertices connected by edges. In other words, there are three connected components in this graph.

The use case of users in a social networking application in which they follow each other is taken up here again for elucidation purposes. By extracting the connected components of the graph, it is possible to see whether any two users are connected or not. *Figure 7* shows the user graph:
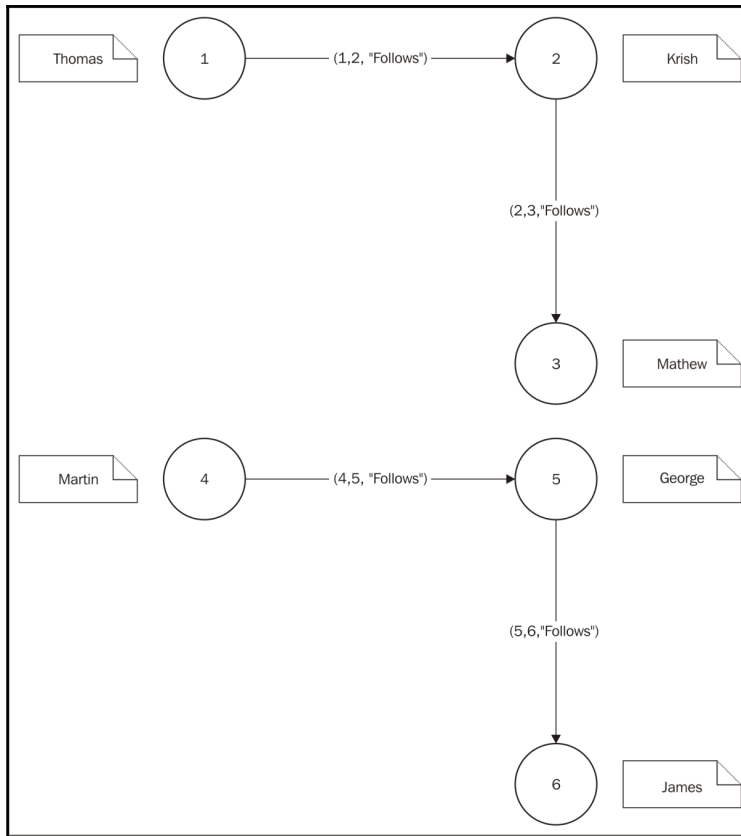
Figure 7

In the graph depicted in *Figure 7*, it is clearly evident that there are two connected components. It is easy to say that Thomas and Mathew are connected and at the same time Thomas and Martin are not connected. If the connected component graph is extracted, it can be seen that Thomas and Martin will have the same connected component identifier, and at the same time, Thomas and Martin will have a different connected component identifiers. At the Scala REPL prompt, try the following statements:

```scala
scala> import org.apache.spark._
   import org.apache.spark._
scala> import org.apache.spark.graphx._
   import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
   import org.apache.spark.rdd.RDD
scala> // Create the RDD with users as the vertices
scala> val users: RDD[(Long, String)] = sc.parallelize(Array((1L,
"Thomas"), (2L, "Krish"),(3L, "Mathew"), (4L, "Martin"), (5L, "George"),
```

```
    (6L, "James")))
users: org.apache.spark.rdd.RDD[(Long, String)] =
ParallelCollectionRDD[1194] at parallelize at <console>:69
scala> // Create the edges connecting the users
scala> val userRelationships: RDD[Edge[String]] =
sc.parallelize(Array(Edge(1L, 2L, "Follows"),Edge(2L, 3L, "Follows"),
Edge(4L, 5L, "Follows"), Edge(5L, 6L, "Follows")))
userRelationships:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[1195] at parallelize at <console>:69
scala> // Create a graph
scala> val userGraph = Graph(users, userRelationships)
userGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@805e363
scala> // Find the connected components of the graph
scala> val cc = userGraph.connectedComponents()
cc: org.apache.spark.graphx.Graph[org.apache.spark.graphx.VertexId,String]
= org.apache.spark.graphx.impl.GraphImpl@13f4a9a9
scala> // Extract the triplets of the connected components
scala> val ccTriplets = cc.triplets
ccTriplets:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.EdgeTriplet[org.apache.spa
rk.graphx.VertexId,String]] = MapPartitionsRDD[1263] at mapPartitions at
GraphImpl.scala:48
scala> // Print the structure of the triplets
scala> ccTriplets.foreach(println)
      ((1,1),(2,1),Follows)
      ((4,4),(5,4),Follows)
      ((5,4),(6,4),Follows)
      ((2,1),(3,1),Follows)
scala> //Print the vertex numbers and the corresponding connected component
id. The connected component id is generated by the system and it is to be
taken only as a unique identifier for the connected component
scala> val ccProperties = ccTriplets.map(triplet => "Vertex " +
triplet.srcId + " and " + triplet.dstId + " are part of the CC with id " +
triplet.srcAttr)
ccProperties: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1264] at
map at <console>:79
scala> ccProperties.foreach(println)
      Vertex 1 and 2 are part of the CC with id 1
      Vertex 5 and 6 are part of the CC with id 4
      Vertex 2 and 3 are part of the CC with id 1
      Vertex 4 and 5 are part of the CC with id 4
scala> //Find the users in the source vertex with their CC id
scala> val srcUsersAndTheirCC = ccTriplets.map(triplet => (triplet.srcId,
triplet.srcAttr))
srcUsersAndTheirCC:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
```

```
org.apache.spark.graphx.VertexId)] = MapPartitionsRDD[1265] at map at
<console>:79
scala> //Find the users in the destination vertex with their CC id
scala> val dstUsersAndTheirCC = ccTriplets.map(triplet => (triplet.dstId,
triplet.dstAttr))
dstUsersAndTheirCC:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
org.apache.spark.graphx.VertexId)] = MapPartitionsRDD[1266] at map at
<console>:79
scala> //Find the union
scala> val usersAndTheirCC = srcUsersAndTheirCC.union(dstUsersAndTheirCC)
usersAndTheirCC:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
org.apache.spark.graphx.VertexId)] = UnionRDD[1267] at union at
<console>:83
scala> //Join with the name of the users
scala> val usersAndTheirCCWithName = usersAndTheirCC.join(users).map{case
(userId,(ccId,userName)) => (ccId, userName)}.distinct.sortByKey()
usersAndTheirCCWithName:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId, String)] =
ShuffledRDD[1277] at sortByKey at <console>:85
scala> //Print the user names with their CC component id. If two users
share the same CC id, then they are connected
scala> usersAndTheirCCWithName.collect().foreach(println)
      (1,Thomas)
      (1,Mathew)
      (1,Krish)
      (4,Martin)
      (4,James)
      (4,George)
```

There are some more graph processing algorithms available in the Spark GraphX library, and a detailed treatment of the complete set of algorithms deserves book on its own. The point here is that the Spark GraphX library provides very easy-to-use graph algorithms that fit very well into Spark's uniform programming model.

Understanding GraphFrames

The Spark GraphX library is the graph processing library that has the least programming language support. Scala is the only programming language supported by the Spark GraphX library. GraphFrames is a new graph processing library available as an external Spark package developed by Databricks, University of California, Berkley, and Massachusetts Institute of Technology, built on top of Spark DataFrames. Since it is built on top of DataFrames, all the operations that can be done on DataFrames are potentially possible on GraphFrames, with support for programming languages such as Scala, Java, Python, and R with a uniform API. Since GraphFrames is built on top of DataFrames, the persistence of data, support for numerous data sources, and powerful graph queries in Spark SQL are additional benefits users get for free.

Just like the Spark GraphX library, in GraphFrames the data is stored in vertices and edges. The vertices and edges use DataFrames as the data structure. The first use case covered in the beginning of this chapter is used again to elucidate GraphFrames-based graph processing.

> **CAUTION**: GraphFrames is an external Spark package. It has some incompatibility with Spark 2.0. Because of that, the following code snippets will not work with Spark 2.0. They work with Spark 1.6. Refer to their website to check Spark 2.0 support.

At the Scala REPL prompt of Spark 1.6, try the following statements. Since GraphFrames is an external Spark package, while bringing up the appropriate REPL, the library has to be imported and the following command is used in the terminal prompt to fire up the REPL and make sure that the library is loaded without any error messages:

```
$ cd $SPARK_1.6__HOME
$ ./bin/spark-shell --packages graphframes:graphframes:0.1.0-spark1.6
Ivy Default Cache set to: /Users/RajT/.ivy2/cache
The jars for the packages stored in: /Users/RajT/.ivy2/jars
:: loading settings :: url = jar:file:/Users/RajT/source-code/spark-
source/spark-1.6.1/assembly/target/scala-2.10/spark-assembly-1.6.2-
SNAPSHOT-hadoop2.2.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
graphframes#graphframes added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0
   confs: [default]
   found graphframes#graphframes;0.1.0-spark1.6 in list
:: resolution report :: resolve 153ms :: artifacts dl 2ms
   :: modules in use:
   graphframes#graphframes;0.1.0-spark1.6 from list in [default]
   -------------------------------------------------------------------
   |                    |            modules            ||   artifacts   |
   |       conf         | number| search|dwnlded|evicted|| number|dwnlded|
```

```
    ------------------------------------------------------------------
    |       default      |   1   |   0   |   0   |   0   ||   1   |   0   |
    ------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent
   confs: [default]
   0 artifacts copied, 1 already retrieved (0kB/5ms)
16/07/31 09:22:11 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Welcome to
          ____              __
         / __/__  ___ _____/ /__
        _\ \/ _ \/ _ `/ __/  '_/
       /___/ .__/\_,_/_/ /_/\_\   version 1.6.1
          /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_66)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.
scala> import org.graphframes._
import org.graphframes._
scala> import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._
scala> //Create a DataFrame of users containing tuple values with a
mandatory Long and another String type as the property of the vertex
scala> val users = sqlContext.createDataFrame(List((1L, "Thomas"),(2L,
"Krish"),(3L, "Mathew"))).toDF("id", "name")
users: org.apache.spark.sql.DataFrame = [id: bigint, name: string]
scala> //Created a DataFrame for Edge with String type as the property of
the edge
scala> val userRelationships = sqlContext.createDataFrame(List((1L, 2L,
"Follows"),(1L, 2L, "Son"),(2L, 3L, "Follows"))).toDF("src", "dst",
"relationship")
userRelationships: org.apache.spark.sql.DataFrame = [src: bigint, dst:
bigint, relationship: string]
scala> val userGraph = GraphFrame(users, userRelationships)
userGraph: org.graphframes.GraphFrame = GraphFrame(v:[id: bigint, name:
string], e:[src: bigint, dst: bigint, relationship: string])
scala> // Vertices in the graph
scala> userGraph.vertices.show()
+---+------+
| id|  name|
```

```
+---+------+
|  1|Thomas|
|  2| Krish|
|  3|Mathew|
+---+------+
scala> // Edges in the graph
scala> userGraph.edges.show()
+---+---+------------+
|src|dst|relationship|
+---+---+------------+
|  1|  2|     Follows|
|  1|  2|         Son|
|  2|  3|     Follows|
+---+---+------------+
scala> //Number of edges in the graph
scala> val edgeCount = userGraph.edges.count()
edgeCount: Long = 3
scala> //Number of vertices in the graph
scala> val vertexCount = userGraph.vertices.count()
vertexCount: Long = 3
scala> //Number of edges coming to each of the vertex.
scala> userGraph.inDegrees.show()
+---+--------+
| id|inDegree|
+---+--------+
|  2|       2|
|  3|       1|
+---+--------+
scala> //Number of edges going out of each of the vertex.
scala> userGraph.outDegrees.show()
+---+---------+
| id|outDegree|
+---+---------+
|  1|        2|
|  2|        1|
+---+---------+
scala> //Total number of edges coming in and going out of each vertex.
scala> userGraph.degrees.show()
+---+------+
| id|degree|
+---+------+
|  1|     2|
|  2|     3|
|  3|     1|
+---+------+
scala> //Get the triplets of the graph
scala> userGraph.triplets.show()
+-------------+----------+----------+
```

```
|         edge|        src|        dst|
+-------------+----------+----------+
|[1,2,Follows]|[1,Thomas]| [2,Krish]|
|     [1,2,Son]|[1,Thomas]| [2,Krish]|
|[2,3,Follows]| [2,Krish]|[3,Mathew]|
+-------------+----------+----------+
scala> //Using the DataFrame API, apply filter and select only the needed
edges
scala> val numFollows = userGraph.edges.filter("relationship =
'Follows'").count()
numFollows: Long = 2
scala> //Create an RDD of users containing tuple values with a mandatory
Long and another String type as the property of the vertex
scala> val usersRDD: RDD[(Long, String)] = sc.parallelize(Array((1L,
"Thomas"), (2L, "Krish"),(3L, "Mathew")))
usersRDD: org.apache.spark.rdd.RDD[(Long, String)] =
ParallelCollectionRDD[54] at parallelize at <console>:35
scala> //Created an RDD of Edge type with String type as the property of
the edge
scala> val userRelationshipsRDD: RDD[Edge[String]] =
sc.parallelize(Array(Edge(1L, 2L, "Follows"),     Edge(1L, 2L,
"Son"),Edge(2L, 3L, "Follows")))
userRelationshipsRDD:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
ParallelCollectionRDD[55] at parallelize at <console>:35
scala> //Create a graph containing the vertex and edge RDDs as created
before
scala> val userGraphXFromRDD = Graph(usersRDD, userRelationshipsRDD)
userGraphXFromRDD: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@77a3c614
scala> //Create the GraphFrame based graph from Spark GraphX based graph
scala> val userGraphFrameFromGraphX: GraphFrame =
GraphFrame.fromGraphX(userGraphXFromRDD)
userGraphFrameFromGraphX: org.graphframes.GraphFrame = GraphFrame(v:[id:
bigint, attr: string], e:[src: bigint, dst: bigint, attr: string])
scala> userGraphFrameFromGraphX.triplets.show()
+-------------+----------+----------+
|         edge|        src|        dst|
+-------------+----------+----------+
|[1,2,Follows]|[1,Thomas]| [2,Krish]|
|     [1,2,Son]|[1,Thomas]| [2,Krish]|
|[2,3,Follows]| [2,Krish]|[3,Mathew]|
+-------------+----------+----------+
scala> // Convert the GraphFrame based graph to a Spark GraphX based graph
scala> val userGraphXFromGraphFrame: Graph[Row, Row] =
userGraphFrameFromGraphX.toGraphX
userGraphXFromGraphFrame:
org.apache.spark.graphx.Graph[org.apache.spark.sql.Row,org.apache.spark.sql
```

```
.Row] = org.apache.spark.graphx.impl.GraphImpl@238d6aa2
```

When creating DataFrames for the GraphFrame, the only thing to keep in mind is that there are some mandatory columns for the vertices and the edges. In the DataFrame for vertices, the id column is mandatory. In the DataFrame for edges, the src and dst columns are mandatory. Apart from that, any number of arbitrary columns can be stored with both the vertices and the edges of a GraphFrame. In the Spark GraphX library, the vertex identifier must be a long integer, but the GraphFrame doesn't have any such limitations and any type is supported as the vertex identifier. Readers should already be familiar with DataFrames; any operation that can be done on a DataFrame can be done on the vertices and edges of a GraphFrame.

> All the graph processing algorithms supported by Spark GraphX are supported by GraphFrames as well.

The Python version of GraphFrames has fewer features. Since Python is not a supported programming language for the Spark GraphX library, GraphFrame to GraphX and GraphX to GraphFrame conversions are not supported in Python. Since readers are familiar with the creation of DataFrames in Spark using Python, the Python example is omitted here. Moreover, there are some pending defects in the GraphFrames API for Python and not all the features demonstrated previously using Scala function properly in Python at the time of writing.

Understanding GraphFrames queries

The Spark GraphX library is the RDD-based graph processing library, but GraphFrames is a Spark DataFrame-based graph processing library that is available as an external package. Spark GraphX supports many graph processing algorithms, but GraphFrames supports not only graph processing algorithms, but also graph queries. The major difference between graph processing algorithms and graph queries is that graph processing algorithms are used to process the data hidden in a graph data structure, while graph queries are used to search for patterns in the data hidden in a graph data structure. In GraphFrame parlance, graph queries are also known as motif finding. This has tremendous applications in genetics and other biological sciences that deal with sequence motifs.

From a use case perspective, take the use case of users following each other in a social media application. Users have relationships between them. In the previous sections, these relationships were modeled as graphs. In real-world use cases, such graphs can become really huge, and if there is a need to find users with relationships between them in both directions, it can be expressed as a pattern in graph query, and such relationships can be found using easy programmatic constructs. The following demonstration models the relationship between the users in a GraphFrame, and a pattern search is done using that.

At the Scala REPL prompt of Spark 1.6, try the following statements:

```
$ cd $SPARK_1.6_HOME
$ ./bin/spark-shell --packages graphframes:graphframes:0.1.0-spark1.6
Ivy Default Cache set to: /Users/RajT/.ivy2/cache
The jars for the packages stored in: /Users/RajT/.ivy2/jars
:: loading settings :: url = jar:file:/Users/RajT/source-code/spark-
source/spark-1.6.1/assembly/target/scala-2.10/spark-assembly-1.6.2-
SNAPSHOT-hadoop2.2.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
graphframes#graphframes added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent;1.0
   confs: [default]
   found graphframes#graphframes;0.1.0-spark1.6 in list
:: resolution report :: resolve 145ms :: artifacts dl 2ms
   :: modules in use:
   graphframes#graphframes;0.1.0-spark1.6 from list in [default]
   ---------------------------------------------------------------------
   |                     |            modules      ||   artifacts   |
   |       conf          | number| search|dwnlded|evicted|| number|dwnlded|
   ---------------------------------------------------------------------
   |      default        |   1   |   0   |   0   |   0   ||   1   |   0   |
   ---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent
   confs: [default]
   0 artifacts copied, 1 already retrieved (0kB/5ms)
16/07/29 07:09:08 WARN NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.1
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java
1.8.0_66)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
```

```
SQL context available as sqlContext.
scala> import org.graphframes._
import org.graphframes._
scala> import org.apache.spark.rdd.RDD
import org.apache.spark.rdd.RDD
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row
scala> import org.apache.spark.graphx._
import org.apache.spark.graphx._
scala> //Create a DataFrame of users containing tuple values with a
mandatory String field as id and another String type as the property of the
vertex. Here it can be seen that the vertex identifier is no longer a long
integer.
scala> val users = sqlContext.createDataFrame(List(("1", "Thomas"),("2",
"Krish"),("3", "Mathew"))).toDF("id", "name")
users: org.apache.spark.sql.DataFrame = [id: string, name: string]
scala> //Create a DataFrame for Edge with String type as the property of
the edge
scala> val userRelationships = sqlContext.createDataFrame(List(("1", "2",
"Follows"),("2", "1", "Follows"),("2", "3", "Follows"))).toDF("src", "dst",
"relationship")
userRelationships: org.apache.spark.sql.DataFrame = [src: string, dst:
string, relationship: string]
scala> //Create the GraphFrame
scala> val userGraph = GraphFrame(users, userRelationships)
userGraph: org.graphframes.GraphFrame = GraphFrame(v:[id: string, name:
string], e:[src: string, dst: string, relationship: string])
scala> // Search for pairs of users who are following each other
scala> // In other words the query can be read like this. Find the list of
users having a pattern such that user u1 is related to user u2 using the
edge e1 and user u2 is related to the user u1 using the edge e2. When a
query is formed like this, the result will list with columns u1, u2, e1 and
e2. When modelling real-world use cases, more meaningful variables can be
used suitable for the use case.
scala> val graphQuery = userGraph.find("(u1)-[e1]->(u2); (u2)-[e2]->(u1)")
graphQuery: org.apache.spark.sql.DataFrame = [e1:
struct<src:string,dst:string,relationship:string>, u1:
struct<id:string,name:string>, u2: struct<id:string,name:string>, e2:
struct<src:string,dst:string,relationship:string>]
scala> graphQuery.show()
+-------------+----------+----------+-------------+
|           e1|        u1|        u2|           e2|
+-------------+----------+----------+-------------+
|[1,2,Follows]|[1,Thomas]| [2,Krish]|[2,1,Follows]|
|[2,1,Follows]| [2,Krish]|[1,Thomas]|[1,2,Follows]|
+-------------+----------+----------+-------------+
```

Note that the columns in the graph query result are formed with the elements given in the search pattern. There is no limit to the way the patterns can be formed.

> Note the data type of the graph query result. It is a DataFrame object. That brings a great flexibility in processing the query results using the familiar Spark SQL library.

The biggest limitation of the Spark GraphX library is that its API is not currently supported with programming languages such as Python and R. Since GraphFrames is a DataFrame-based library, once it has matured, it will enable graph processing in all the programming languages supported by DataFrames. This Spark external package is definitely a potential candidate to be included as part of the Spark.

References

For more information please visit the following links:

- `https://spark.apache.org/docs/1.5.2/graphx-programming-guide.html`
- `https://en.wikipedia.org/wiki/215_ATP_World_Tour_Finals_%E2%8%93_Singles`
- `http://www.protennislive.com/posting/215/65/mds.pdf`
- `http://infolab.stanford.edu/~backrub/google.html`
- `http://graphframes.github.io/index.html`
- `https://github.com/graphframes/graphframes`
- `https://spark-packages.org/package/graphframes/graphframes`

Summary

A Graph is a very useful data structure that has great application potential. Even though it is not very commonly used in most applications, there are some unique application use cases where using a Graph as a data structure is essential. A data structure is effectively used only when it is used in conjunction with well tested and highly optimized algorithms. Mathematicians and computer scientists have come up with many algorithms to process data that is part of a graph data structure. The Spark GraphX library has a large number of such algorithms implemented on top of the Spark core. This chapter provided a whirlwind tour of the Spark GraphX library and covered some of the basics through use cases at an introductory level.

The DataFrame-based graph abstraction named GraphFrames, which comes in an external Spark package available separately from Spark, has tremendous potential in graph processing as well as graph queries. A brief introduction to this external Spark package has been provided in order to do graph queries to find patterns in graphs.

Any book teaching a new technology has to conclude with an application covering its salient features. Spark is no different. So far in this book, Spark as a next generation data processing platform has been covered. Now it is the time to tie up all the loose ends and build an end-to-end application. The next chapter is going to cover the design and development of a data processing application using Spark and the family of libraries built on top of it.

9
Designing Spark Applications

Think functionally. Think application functionality designed like a pipeline with each piece plumbed together doing some part of the whole job in hand. It is all about processing data, and that is what Spark does in a highly versatile manner. Data processing starts with the seed data that gets into the processing pipeline. The seed data can be a new piece of data that is ingested into the system, or it can be some kind of master dataset that lives in the enterprise data store and needs to be sliced and diced to produce different views to serve various purposes and business needs. It is this slicing and dicing that is going to be the norm when designing and developing data processing applications.

Any application development exercise starts with a study of the domain, the business requirements, and the technical tool selection. It is not going to be different here. Even though this chapter is going to see the design and development of a Spark application, the initial focus will be on the overall architecture of data processing applications, use cases, the data, and the applications that transform the data from one state to another. Spark is just a driver that assembles data processing logic and data together, using its highly powerful infrastructure to produce the desired results.

We will cover the following topics in this chapter:

- Lambda Architecture
- Microblogging with Spark
- Data dictionaries
- Coding style
- Data ingestion

Lambda Architecture

Application architecture is very important for any kind of software development. It is the blueprint that decides how the software has to be built with a good amount of generality and the capability to customize when needed. For common application needs, some popular architectures are available, and there is no need for any ground-up architecture effort in order to use them. These public architecture frameworks are designed by some of the best minds for the benefit of the general public. These popular architectures are very useful because there is no barrier to entry, and they are used by so many people. There are popular architectures available for web application development, data processing, and so on.

Lambda Architecture is a recent and popular architecture that's ideal for developing data processing applications. There are many tools and technologies available in the market to develop data processing applications. But independent of the technology, how the data processing application components are layered and composed together is driven by the architectural framework. That is why Lambda Architecture is a technology-agnostic architecture framework and, depending on the need, the appropriate technology choice can be made to develop the individual components. *Figure 1* captures the essence of Lambda Architecture:
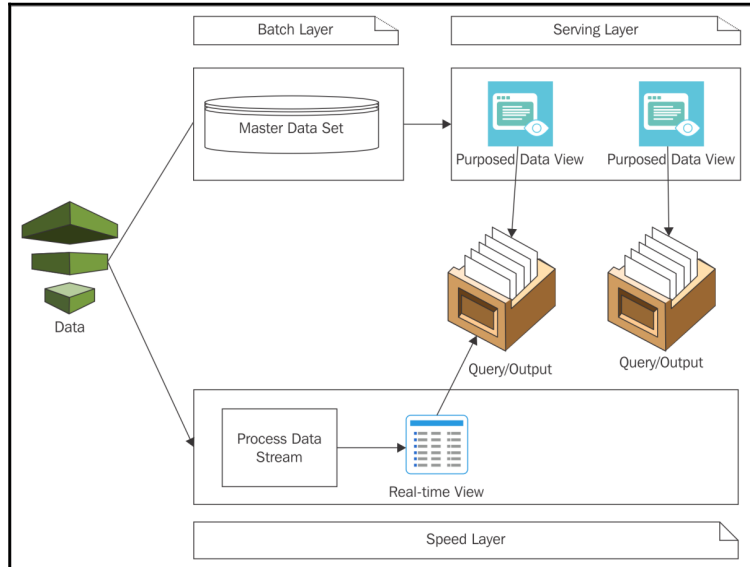
Figure 1

Lambda Architecture consists of three layers:

- The batch layer is the main data store. Any kind of processing happens on this dataset. This is the golden dataset.
- The serving layer processes the master dataset and prepares views for specific purposes, and they are termed purposed views here. This intermediate step of processing is required to serve the queries, or for generating outputs for specific needs. The queries and the specific dataset preparation don't directly access the master dataset.
- The speed layer is all about data stream processing. The stream of data is processed in a real-time fashion and volatile real-time views are prepared if that is a business need. The queries or specific processes generating outputs may consume data from both the purposed data views and real-time views.

Using the principles of Lambda Architecture to architect a big data processing system, Spark is going to be used here as a data processing tool. Spark fits nicely into all the data processing requirements in all three distinct layers.

This chapter is going to discuss some selected data processing use cases of a microblogging application. The application functionality, its deployment infrastructure, and the scalability factors are beyond the scope of this work. In a typical batch layer, the master dataset can be plain splittable serialization formats or NoSQL data stores, depending on the data access methods. If the application use cases are all batch operations, then standard serialization formats will be sufficient. But if the use cases mandate random access, NoSQL data stores will be ideal. Here, for the sake of simplicity, all the data files are stored in plain text files locally.

Typical application development culminates in a completely functional application. But here, the use cases are realized in Spark data processing applications. Data processing always works as a part of the functionality of the main application and it is scheduled to run in batch mode or run as a listener waiting for data and processing it. So, corresponding to each of the use cases, individual Spark applications are developed, and they can be scheduled or made to run in listener mode as the case may be.

Microblogging with Lambda Architecture

Blogging has been around for couple of decades in various forms. In the initial days of blogging as a medium of publication, only professional or aspiring writers published articles through the medium of blogs. It spread the false notion that only serious content is published through blogs. In recent years, the concept of microblogging included the general public in the culture of blogging. Microblogs are sudden outbursts of the thought processes of people in the form of a few sentences, photos, videos, or links. Sites such as Twitter and Tumblr popularized this culture at the biggest scale possible with hundreds of millions of active users using the site.

An overview of SfbMicroBlog

SfbMicroBlog is a microblogging application with millions of users posting short messages. A new user who is going to use this application needs to sign up with a username and password. To post messages, users have to sign in first. The only thing users can do without signing in is read public messages posted by users. Users can follow other users. The act of following is a one-way relationship. If user A follows user B, user A can see all the messages posted by user B; at the same time, user B cannot see the messages posted by user A, because user B is not following user A. By default, all the messages posted by all the users are public messages and can be seen by everybody. But users have settings to make messages visible only to users who are following the message owner. After becoming a follower, unfollowing is also allowed.

Usernames have to be unique across all users. A username and password are required to sign in. Every user must have a primary e-mail address, and without that the signup process will not be complete. For extra security and password recovery, an alternate e-mail address or mobile phone number can be saved in the profile.

Messages cannot exceed the a of 140 characters. Messages can contain words prefixed with the # symbol to group them under various topics. Messages can contain usernames prefixed with the @ symbol to directly address users through messages that are posted. In other words, users can address any other user in their messages without being a follower.

Once posted, the messages cannot be changed. Once posted, the messages cannot be deleted.

Getting familiar with data

All pieces of data that come to the master dataset come through a stream. The data stream is processed, an appropriate header for each message is inspected, and the right action to store it in the data store is done. The following list contains the important data items that come into the store through the same stream:

- **User**: This dataset contains the user details when a user signs in or when a user's data gets changed
- **Follower**: This dataset contains the relationship data that gets captured when a user opts to follow another user
- **Message**: This dataset contains the messages posted by registered users

This list of datasets forms the golden dataset. Based on this master dataset, various views are created that cater to the needs of the vital business functions in the application. The following list contains the important views of the master dataset:

- **Messages by users**: This view contains messages posted by each user in the system. When a given user wants to see the messages posted by him/her, the data generated by this view is used. This is also used by the given user's followers. This is a situation where the main dataset is used for a specific purpose. The message dataset gives all the required data for this view.
- **Messages to users**: In the messages, specific users can be addressed by prefixing the @ symbol followed by the addressee's username. This data view contains the users addressed with the @ symbol and the corresponding messages. There is a limitation enforced in the implementation: you can only have one addressee in one message.
- **Tagged messages**: In the messages, words prefixed with the # symbol becomes searchable messages. For example, the word #spark in a message signifies that the message is searchable by the word #spark. For a given hashtag, users can see all the public messages and the messages of users whom he/she is following in one list. This view contains pairs of the hashtag and the corresponding messages. There is a limitation enforced in the implementation: you can only have one tag in one message.
- **Follower users**: This view contains the list of users who are following a given user. In *Figure 2*, users **U1** and **U3** are in the list of users following **U4**.
- **Followed users**: This view contains the list of users who are followed by a given user. In *Figure 2*, users **U2** and **U4** are in the list of users who are followed by user **U1**:
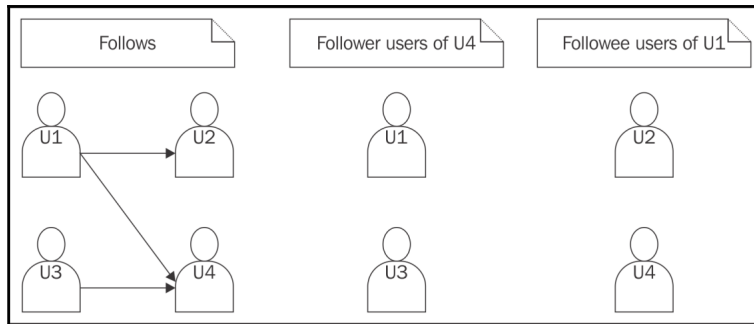
Figure 2

In a nutshell, *Figure 3* gives the Lambda Architecture view of the solution and gives details of the datasets and the corresponding views:
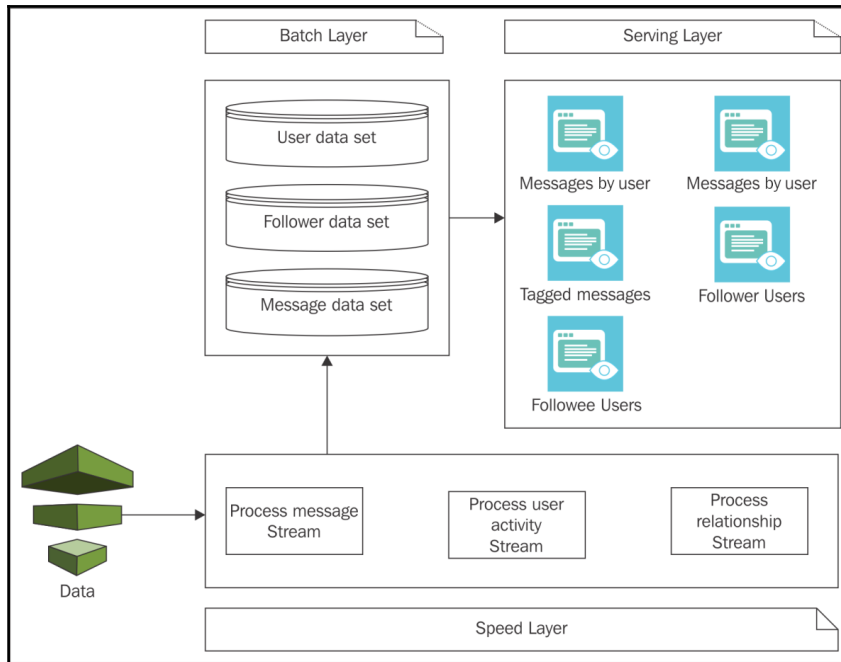
Figure 3

Setting the data dictionary

The data dictionary describes the data, its meaning, and its relationship with other data items. For the SfbMicroBlog application, the data dictionary is going to be a very minimalistic one to implement the selected use cases. Using this as a base, readers can expand and implement their own data items and include data processing use cases. The data dictionary is given for all the master datasets, as well as the data views.

The following table shows the data items of the user dataset:

User data	Type	Purpose
Id	Long	Used to uniquely identify a user, as well as being the vertex identifier in the user relationship graph
Username	String	Used to uniquely identify users of the system
First name	String	Used to capture the first name of the user
Last name	String	Used to capture the last name of the user
E-mail	String	Used to communicate with users
Alternate e-mail	String	Used for password recovery
Primary phone	String	Used for password recovery

The following table captures the data items of the follower dataset:

Follower data	Type	Purpose
Follower username	String	Used to identify who the follower is
Followed username	String	Used to identify who is being followed

The following table captures the data items of the message dataset:

Message data	Type	Purpose
Username	String	Used to capture the user who posted the message
Message Id	Long	Used to uniquely identify a message
Message	String	Used to capture the message that is being posted
Timestamp	Long	Used to capture the time at which the message is posted

The following table captures the data items of the Message to users view:

Message to users data	Type	Purpose
From username	String	Used to capture the user who posted the message
To username	String	Used to capture the user to whom the message is addressed; it is the username that is prefixed with the @ symbol
Message Id	Long	Used to uniquely identify a message
Message	String	Used to capture the message that is being posted
Timestamp	Long	Used to capture the time at which the message is posted

The following table captures the data items of the Tagged messages view:

Tagged messages data	Type	Purpose
Hashtag	String	The word that is prefixed with the # symbol
Username	String	Used to capture the user who posted the message
Message Id	Long	Used to uniquely identify a message
Message	String	Used to capture the message that is being posted
Timestamp	Long	Used to capture the time at which the message is posted

The follower relationship of the users is pretty straightforward and consists of the pairs of user identification numbers persisted in a data store.

Implementing Lambda Architecture

The concept of Lambda Architecture was introduced in the beginning of this chapter. Since it is a technology-agnostic architecture framework, when designing applications with it, it is imperative to capture the technology choices used in specific implementations. The following sections do exactly that.

Batch layer

The core of the batch layer is a data store. For big data applications, there are plenty of choices for data stores. Typically, **Hadoop Distributed File System** (**HDFS**) in conjunction with Hadoop YARN is the current and accepted platform in which data is stored, mainly because of the ability to partition and distribute data across the Hadoop cluster.

There are two types of data access any persistence store supports:

- Batch write/read
- Random write/read

Both of these need separate data storage solutions. For batch data operations, typically splittable serialization formats such as Avro and Parquet are used. For random data operations, typically NoSQL data stores are used. Some of these NoSQL solutions sit on top of HDFS and some don't. It doesn't matter whether they are on top of HDFS or not, they provide partitioning and distribution of data. So depending on the use case and the distributed platform that is in use, appropriate solutions can be used.

When it comes to the storage of the data in HDFS, commonly used formats such as XML and JSON fail because HDFS partitions and distributes the files. When that happens, these formats have opening tags and ending tags, and splits at random locations in the file make the data dirty. Because of that, splittable file formats such as Avro or Parquet are efficient for storing in HDFS.

When it comes to the NoSQL data store solutions, there are many choices in the market, especially from the open source world. Some of these NoSQL data stores, such as Hbase, sit on top of HDFS. Some of the NoSQL data stores, such as Cassandra and Riak, don't need HDFS, can be deployed on regular operating systems, and can be deployed in master-less fashion so that there is no single point of failure in a cluster. The choice of the NoSQL store is again dependent on the usage of a particular technology within the organization, the production support contracts in place, and many other parameters.

> This book doesn't recommend a given set of data store technologies for usage in conjunction with Spark because Spark drivers are available in abundance for most popular serialization formats and NoSQL data stores. In other words, most of the data store vendors have started supporting Spark in big way. Another interesting trend these days is that many of the prominent ETL tools have started supporting Spark, and because of that, those who are using such ETL tools may use Spark applications within their ETL processing pipelines.

In this application, neither an HDFS-based nor any NoSQL-based data store is being used in order to maintain simplicity and to avoid the complex infrastructure setup required to run the application for the readers. Throughout, the data is stored on the local system in text file formats. Readers who are interested in trying out the examples on HDFS or other NoSQL data stores may go ahead and try them, with some changes to the data write/read part of the application.

Serving layer

The serving layer can be implemented in Spark using various methods. If the data is not structured and is purely object-based, then the low-level RDD-based method is suitable. If the data is structured, a DataFrame is ideal. The use case that is being discussed here is dealing with structured data and hence wherever possible the Spark SQL library is going to be used. From the data stores, data is read and RDDs are created. The RDDs are converted to DataFrames and all the serving needs are accomplished using Spark SQL. In this way, the code is going to be succinct and easy to understand.

Speed layer

The speed layer is going to be implemented as a Spark Streaming application using Kafka as the broker with its own producers producing the messages. The Spark Streaming application will act as the consumer to the Kafka topics and receive the data that is getting produced. As discussed in the chapter covering Spark Streaming, the producers can be the Kafka console producer or any other producer supported by Kafka. But the Spark Streaming application working as the consumer here is not going to implement the logic of persisting the processed messages to the text file as they are not generally used in real-world use cases. Using this application as a base, readers can implement their own persistence mechanism.

Queries

The queries are all generated from the speed layer and serving layer. Since the data is available in the form of DataFrames, as mentioned before, all the queries for the use case are implemented using Spark SQL. The obvious reason is that Spark SQL works as a consolidation technology that unifies the data sources and destinations. When readers are using the samples from this book and when they are ready to implement it in their real-world use cases, the overall methodology can remain the same, but the data sources and destinations may differ. The following are some of the queries that can be generated from the serving layer. It is up to the imagination of the readers to make the required changes to the data dictionary and be able to write these views or queries:

- Find the messages that are grouped by a given hashtag
- Find the messages that are addressed to a given user
- Find the followers of a given user
- Find the followed users of a given user

Working with Spark applications

The workhorse of this application is the data processing engine consisting of many Spark applications. In general, they can be classified into the following types:

- A Spark Streaming application to ingest data: This is the main listener application that receives the data coming as a stream and stores it in the appropriate master dataset.
- A Spark application to create purposed views and queries: This is the application that is used to create various purposed views from the master datasets. Apart from that, the queries are also included in this application.
- A Spark GraphX application to do custom data processing: This is the application that is used to process the user-follower relationship.

All these applications are developed independently and they are submitted independently, but the stream processing application will be always running as a listener application to process the incoming messages. Apart from the main data streaming application, all the other applications are scheduled like regular jobs, such as cron jobs in a UNIX system. In this application, all these applications are producing various purposed views. The scheduling depends on the kind of application and how much delay is affordable between the main dataset and the views. It completely depends on the business functions. So this chapter is going to focus on Spark application development rather than scheduling, to keep the focus on the lessons learned in the earlier chapters.

> It is not ideal to persist the data from the speed layer into text files when implementing real-world use cases. For simplicity, all the data is stored in text files to empower all levels of reader with the simplest setup. The speed layer implementation using Spark Streaming is a skeleton implementation without the persistence logic. Readers can enhance this to introduce persistence to their desired data stores.

Coding style

Coding style has been discussed and lots of Spark application programming has been done in the earlier chapters. By now, it has been proven in this book that Spark application development can be done in Scala, Python, and R. In most of the earlier chapters, the languages of choice were Scala and Python. In this chapter, the same trend will continue. Only for the Spark GraphX application, since there is no Python support, will the application be developed in Scala alone.

The style of coding is going to be simple and to the point. The error handling and other best practices of application development are avoided deliberately to focus on the Spark features. In this chapter, wherever possible, the code is run from the appropriate language's Spark REPL. Since the anatomy of the complete application and the scripts to compile, build, and run them as applications have already been covered in the chapter that discussed Spark Streaming, the source code download will have it available as complete ready-to-run applications. Moreover, the chapter covering Spark Streaming discussed the anatomy of a complete Spark application, including the scripts to build and run Spark applications. The same methodology will be used in the applications that are going to be developed in this chapter too. When running such standalone Spark applications, as discussed in the initial chapters of this book, readers can enable Spark monitoring and see how the application is behaving. For the sake of brevity, these discussions will not be taken up again here.

Setting up the source code

Figure 4 shows the structure of the source code and the data directories that are being used in this chapter. A description of each of them is not provided here as the reader should be familiar with them, and they have been covered in `Chapter 6`, *Spark Stream Processing*. There are external library file dependency requirements for running the programs using Kafka. For that, the instructions to download the JAR file are in the `TODO.txt` file in the `lib` folders. The `submitPy.sh` and `submit.sh` files use some of the `Kafka` libraries in the Kafta installation as well. All these external JAR file dependencies have already been covered in `Chapter 6`, *Spark Stream Processing*.
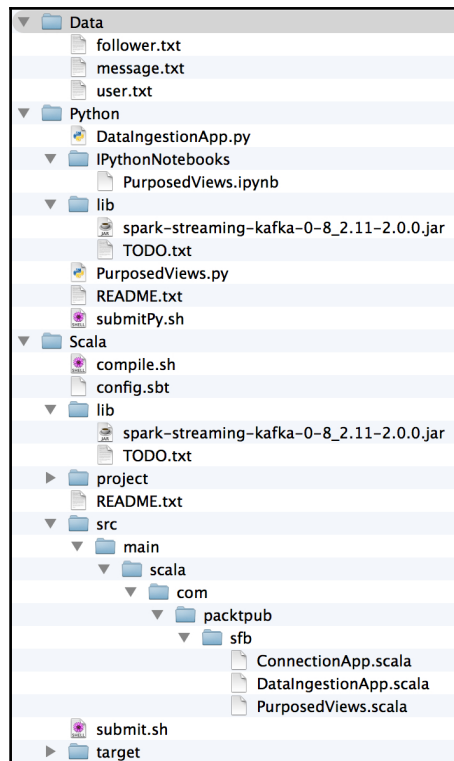
Figure 4

Understanding data ingestion

The Spark Streaming application works as the listener application that receives the data from its producers. Since Kafka is going to be used as the message broker, the Spark Streaming application will be its consumer application, listening to the topics for the messages sent by its producers. Since the master dataset in the batch layer has the following datasets, it is ideal to have individual Kafka topics for each of the topics, along with the datasets.

- User dataset: User
- Follower dataset: Follower
- Message dataset: Message

Figure 5 provides an overall picture of the Kafka-based Spark Streaming application structure:
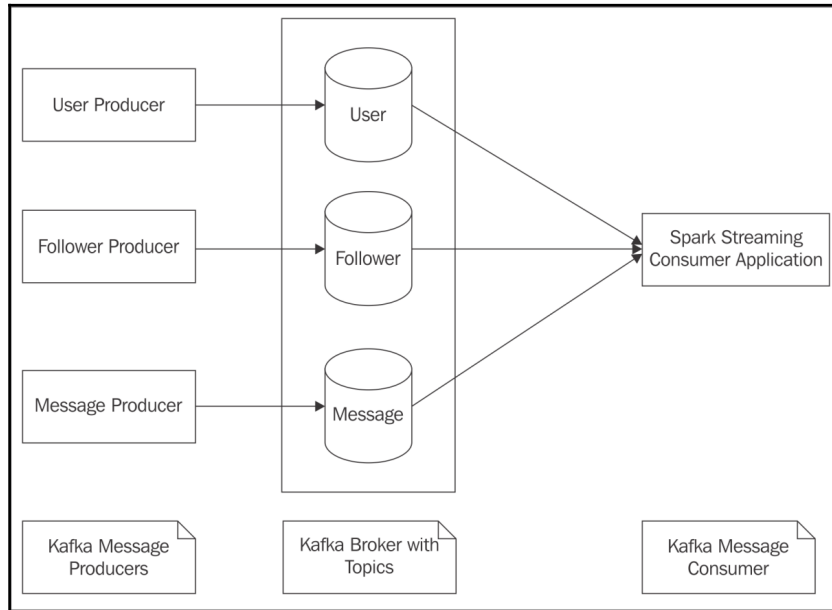
Figure 5

Since the Kafka setup has already been covered in `Chapter 6`, *Spark Stream Processing*, only the application code is covered here.

The following scripts are run from a terminal window. Make sure that the `$KAFKA_HOME` environment variable is pointing to the directory where Kafka is installed. Also, it is very important to start Zookeeper, the Kafka server, the Kafka producer, and the Spark Streaming log event data processing application in separate terminal windows. Once the necessary Kafka topics are created as shown in the scripts, the appropriate producers have to start producing messages. Refer to the Kafka setup details that have already been covered in `Chapter 6`, *Spark Stream Processing*, before proceeding further.

Try the following commands in the terminal window prompt:

```
$ # Start the Zookeeper
$ cd $KAFKA_HOME
$ $KAFKA_HOME/bin/zookeeper-server-start.sh
$KAFKA_HOME/config/zookeeper.properties
    [2016-07-30 12:50:15,896] INFO binding to port 0.0.0.0/0.0.0.0:2181
(org.apache.zookeeper.server.NIOServerCnxnFactory)
$ # Start the Kafka broker in a separate terminal window
$ $KAFKA_HOME/bin/kafka-server-start.sh
$KAFKA_HOME/config/server.properties
    [2016-07-30 12:51:39,206] INFO [Kafka Server 0], started
(kafka.server.KafkaServer)
$ # Create the necessary Kafka topics. This is to be done in a separate
terminal window
$ $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 -
-replication-factor 1 --partitions 1 --topic user
    Created topic "user".
$ $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 -
-replication-factor 1 --partitions 1 --topic follower
    Created topic "follower".
$ $KAFKA_HOME/bin/kafka-topics.sh --create --zookeeper localhost:2181 -
-replication-factor 1 --partitions 1 --topic message
    Created topic "message".
$ # Start producing messages and publish to the topic "message"
$ $KAFKA_HOME/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic message
```

This section provides the details of the Scala code for the Kafka topic consumer application that processes the messages produced by the Kafka producer. The assumption before running the following code snippet is that Kafka is up and running, the required producers are producing messages, and then, if the application is run, it will start consuming the messages. The Scala program for the data ingestion is run by submitting it to the Spark cluster. Starting from the Scala directory, as shown in *Figure 4*, first compile the program and then run it. The README.txt file is to be consulted for additional instructions. The two following commands are to be executed to compile and run the program:

```
$ ./compile.sh
$ ./submit.sh com.packtpub.sfb.DataIngestionApp 1
```

The following code is the program listing that is to be compiled and run using the preceding commands:

```
/**
The following program can be compiled and run using SBT
Wrapper scripts have been provided with this
The following script can be run to compile the code
```

```
    ./compile.sh
    The following script can be used to run this application in Spark. The
second command line argument of value 1 is very important. This is to flag
the shipping of the kafka jar files to the Spark cluster
    ./submit.sh com.packtpub.sfb.DataIngestionApp 1
    **/
    package com.packtpub.sfb
    import java.util.HashMap
    import org.apache.spark.streaming._
    import org.apache.spark.sql.{Row, SparkSession}
    import org.apache.spark.streaming.kafka._
    import org.apache.kafka.clients.producer.{ProducerConfig,
KafkaProducer, ProducerRecord}
    import org.apache.spark.storage.StorageLevel
    import org.apache.log4j.{Level, Logger}
    object DataIngestionApp {
      def main(args: Array[String]) {
       // Log level settings
       LogSettings.setLogLevels()
       //Check point directory for the recovery
       val checkPointDir = "/tmp"
        /**
         * The following function has to be used to have checkpointing and
driver recovery
         * The way it should be used is to use the
StreamingContext.getOrCreate with this function and do a start of that
        * This function example has been discussed but not used in the
chapter covering Spark Streaming. But here it is being used
        */
        def sscCreateFn(): StreamingContext = {
           // Variables used for creating the Kafka stream
           // Zookeeper host
              val zooKeeperQuorum = "localhost"
           // Kaka message group
           val messageGroup = "sfb-consumer-group"
           // Kafka topic where the programming is listening for the data
           // Reader TODO: Here only one topic is included, it can take a
comma separated string containing the list of topics.
           // Reader TODO: When using multiple topics, use your own logic to
extract the right message and persist to its data store
           val topics = "message"
           val numThreads = 1
           // Create the Spark Session, the spark context and the streaming
context
           val spark = SparkSession
               .builder
               .appName(getClass.getSimpleName)
               .getOrCreate()
```

```
            val sc = spark.sparkContext
                val ssc = new StreamingContext(sc, Seconds(10))
                val topicMap = topics.split(",").map((_,
numThreads.toInt)).toMap
                val messageLines = KafkaUtils.createStream(ssc,
zooKeeperQuorum, messageGroup, topicMap).map(_._2)
            // This is where the messages are printed to the console.
            // TODO - As an exercise to the reader, instead of printing
messages to the console, implement your own persistence logic
            messageLines.print()
            //Do checkpointing for the recovery
            ssc.checkpoint(checkPointDir)
            // return the Spark Streaming Context
            ssc
         }
        // Note the function that is defined above for creating the Spark
streaming context is being used here to create the Spark streaming context.
        val ssc = StreamingContext.getOrCreate(checkPointDir, sscCreateFn)
        // Start the streaming
         ssc.start()
        // Wait till the application is terminated
         ssc.awaitTermination()
      }
    }
    object LogSettings {
      /**
       Necessary log4j logging level settings are done
      */
      def setLogLevels() {
        val log4jInitialized =
Logger.getRootLogger.getAllAppenders.hasMoreElements
        if (!log4jInitialized) {
          // This is to make sure that the console is clean from other INFO
messages printed by Spark
          Logger.getRootLogger.setLevel(Level.INFO)
        }
      }
    }
```

The Python program for the data ingestion is run by submitting it to the Spark cluster. Starting from the Python directory, as shown in *Figure 4*, run the program. The README.txt file is to be consulted for additional instructions. All the Kafka installation requirements are valid, even when running this Python program. The following command is to be followed for running the program. Since Python is an interpreted language, there is no compilation required here:

```
$ ./submitPy.sh DataIngestionApp.py 1
```

The following code snippet is the Python implementation of the same application:

```python
# The following script can be used to run this application in Spark
# ./submitPy.sh DataIngestionApp.py 1
  from __future__ import print_function
  import sys
  from pyspark import SparkContext
  from pyspark.streaming import StreamingContext
  from pyspark.streaming.kafka import KafkaUtils
  if __name__ == "__main__":
# Create the Spark context
  sc = SparkContext(appName="DataIngestionApp")
  log4j = sc._jvm.org.apache.log4j
  log4j.LogManager.getRootLogger().setLevel(log4j.Level.WARN)
# Create the Spark Streaming Context with 10 seconds batch interval
  ssc = StreamingContext(sc, 10)
# Check point directory setting
  ssc.checkpoint("\tmp")
# Zookeeper host
  zooKeeperQuorum="localhost"
# Kaka message group
  messageGroup="sfb-consumer-group"
# Kafka topic where the programming is listening for the data
# Reader TODO: Here only one topic is included, it can take a comma
separated  string containing the list of topics.
# Reader TODO: When using multiple topics, use your own logic to extract
the right message and persist to its data store
topics = "message"
numThreads = 1
# Create a Kafka DStream
kafkaStream = KafkaUtils.createStream(ssc, zooKeeperQuorum, messageGroup,
{topics: numThreads})
messageLines = kafkaStream.map(lambda x: x[1])
# This is where the messages are printed to the console. Instead of this,
implement your own persistence logic
messageLines.pprint()
# Start the streaming
ssc.start()
# Wait till the application is terminated
ssc.awaitTermination()
```

Generating purposed views and queries

The following implementations in Scala and Python are the application that creates the purposed views and queries discussed in the earlier sections of this chapter. At the Scala REPL prompt, try the following statements:

```
//TODO: Change the following directory to point to your data directory
scala> val dataDir =
"/Users/RajT/Documents/Writing/SparkForBeginners/To-
PACKTPUB/Contents/B05289-09-DesigningSparkApplications/Code/Data/"
     dataDir: String = /Users/RajT/Documents/Writing/SparkForBeginners/To-
PACKTPUB/Contents/B05289-09-DesigningSparkApplications/Code/Data/
scala> //Define the case classes in Scala for the entities
scala> case class User(Id: Long, UserName: String, FirstName: String,
LastName: String, EMail: String, AlternateEmail: String, Phone: String)
     defined class User
scala> case class Follow(Follower: String, Followed: String)
     defined class Follow
scala> case class Message(UserName: String, MessageId: Long,
ShortMessage: String, Timestamp: Long)
     defined class Message
scala> case class MessageToUsers(FromUserName: String, ToUserName:
String, MessageId: Long, ShortMessage: String, Timestamp: Long)
     defined class MessageToUsers
scala> case class TaggedMessage(HashTag: String, UserName: String,
MessageId: Long, ShortMessage: String, Timestamp: Long)
     defined class TaggedMessage
scala> //Define the utility functions that are to be passed in the
applications
scala> def toUser =  (line: Seq[String]) => User(line(0).toLong,
line(1), line(2),line(3), line(4), line(5), line(6))
     toUser: Seq[String] => User
scala> def toFollow =  (line: Seq[String]) => Follow(line(0), line(1))
     toFollow: Seq[String] => Follow
scala> def toMessage =  (line: Seq[String]) => Message(line(0),
line(1).toLong, line(2), line(3).toLong)
     toMessage: Seq[String] => Message
scala> //Load the user data into a Dataset
scala> val userDataDS = sc.textFile(dataDir +
"user.txt").map(_.split("\\|")).map(toUser(_)).toDS()
     userDataDS: org.apache.spark.sql.Dataset[User] = [Id: bigint,
UserName: string ... 5 more fields]
scala> //Convert the Dataset into data frame
scala> val userDataDF = userDataDS.toDF()
     userDataDF: org.apache.spark.sql.DataFrame = [Id: bigint, UserName:
string ... 5 more fields]
scala> userDataDF.createOrReplaceTempView("user")
```

```
scala> userDataDF.show()
    +---+--------+---------+--------+-------------------+---------------
-+-------------+
    | Id|UserName|FirstName|LastName|              EMail|
AlternateEmail|       Phone|
    +---+--------+---------+--------+-------------------+---------------
-+-------------+
    |  1| mthomas|     Mark|  Thomas|
mthomas@example.com|mt12@example.com|+4411860297701|
    |  2|mithomas|  Michael|  Thomas|mithomas@example.com|
mit@example.com|+4411860297702|
    |  3|  mtwain|     Mark|   Twain|  mtwain@example.com|
mtw@example.com|+4411860297703|
    |  4|  thardy|   Thomas|   Hardy|  thardy@example.com|
th@example.com|+4411860297704|
    |  5| wbryson|  William|  Bryson| wbryson@example.com|
bb@example.com|+4411860297705|
    |  6|   wbrad|  William|Bradford|   wbrad@example.com|
wb@example.com|+4411860297706|
    |  7|  eharris|      Ed|  Harris| eharris@example.com|
eh@example.com|+4411860297707|
    |  8|   tcook|   Thomas|    Cook|   tcook@example.com|
tk@example.com|+4411860297708|
    |  9|  arobert|    Adam|  Robert| arobert@example.com|
ar@example.com|+4411860297709|
    | 10|  jjames|    Jacob|   James|  jjames@example.com|
jj@example.com|+4411860297710|
    +---+--------+---------+--------+-------------------+---------------
-+-------------+
scala> //Load the follower data into an Dataset
scala> val followerDataDS = sc.textFile(dataDir +
"follower.txt").map(_.split("\\|")).map(toFollow(_)).toDS()
    followerDataDS: org.apache.spark.sql.Dataset[Follow] = [Follower:
string, Followed: string]
scala> //Convert the Dataset into data frame
scala> val followerDataDF = followerDataDS.toDF()
    followerDataDF: org.apache.spark.sql.DataFrame = [Follower: string,
Followed: string]
scala> followerDataDF.createOrReplaceTempView("follow")
scala> followerDataDF.show()
    +--------+--------+
    |Follower|Followed|
    +--------+--------+
    | mthomas|mithomas|
    | mthomas|  mtwain|
    |  thardy| wbryson|
    |   wbrad| wbryson|
    | eharris| mthomas|
```

```
      | eharris|   tcook|
      | arobert|  jjames|
      +-------+--------+
    scala> //Load the message data into an Dataset
    scala> val messageDataDS = sc.textFile(dataDir +
  "message.txt").map(_.split("\\|")).map(toMessage(_)).toDS()
      messageDataDS: org.apache.spark.sql.Dataset[Message] = [UserName:
  string, MessageId: bigint ... 2 more fields]
    scala> //Convert the Dataset into data frame
    scala> val messageDataDF = messageDataDS.toDF()
      messageDataDF: org.apache.spark.sql.DataFrame = [UserName: string,
  MessageId: bigint ... 2 more fields]
    scala> messageDataDF.createOrReplaceTempView("message")
    scala> messageDataDF.show()
      +--------+---------+--------------------+----------+
      |UserName|MessageId|        ShortMessage| Timestamp|
      +--------+---------+--------------------+----------+
      | mthomas|        1|@mithomas Your po...|1459009608|
      | mthomas|        2|Feeling awesome t...|1459010608|
      |  mtwain|        3|My namesake in th...|1459010776|
      |  mtwain|        4|Started the day w...|1459011016|
      |  thardy|        5|It is just spring...|1459011199|
      | wbryson|        6|Some days are rea...|1459011256|
      |   wbrad|        7|@wbryson Stuff ha...|1459011333|
      | eharris|        8|Anybody knows goo...|1459011426|
      |   tcook|        9|Stock market is p...|1459011483|
      |   tcook|       10|Dont do day tradi...|1459011539|
      |   tcook|       11|I have never hear...|1459011622|
      |   wbrad|       12|#Barcelona has pl...|1459157132|
      |  mtwain|       13|@wbryson It is go...|1459164906|
      +--------+---------+--------------------+----------+
```

These steps complete the process of loading all the required data from persistent stores into DataFrames. Here, the data comes from text files. In real-world use cases, it may come from popular NoSQL data stores, traditional RDBMS tables, or from Avro or Parquet serialized data stores loaded from HDFS.

The following section uses these DataFrames and creates various purposed views and queries:

```
    scala> //Create the purposed view of the message to users
    scala> val messagetoUsersDS =
  messageDataDS.filter(_.ShortMessage.contains("@")).map(message =>
  (message.ShortMessage.split(" ").filter(_.contains("@")).mkString("
  ").substring(1), message)).map(msgTuple =>
  MessageToUsers(msgTuple._2.UserName, msgTuple._1, msgTuple._2.MessageId,
  msgTuple._2.ShortMessage, msgTuple._2.Timestamp))
      messagetoUsersDS: org.apache.spark.sql.Dataset[MessageToUsers] =
```

```
[FromUserName: string, ToUserName: string ... 3 more fields]
    scala> //Convert the Dataset into data frame
    scala> val messagetoUsersDF = messagetoUsersDS.toDF()
      messagetoUsersDF: org.apache.spark.sql.DataFrame = [FromUserName:
string, ToUserName: string ... 3 more fields]
    scala> messagetoUsersDF.createOrReplaceTempView("messageToUsers")
    scala> messagetoUsersDF.show()
      +------------+---------+---------+--------------------+----------+
      |FromUserName|ToUserName|MessageId|        ShortMessage| Timestamp|
      +------------+---------+---------+--------------------+----------+
      |     mthomas|  mithomas|        1|@mithomas Your po...|1459009608|
      |       wbrad|   wbryson|        7|@wbryson Stuff ha...|1459011333|
      |      mtwain|   wbryson|       13|@wbryson It is go...|1459164906|
      +------------+---------+---------+--------------------+----------+
    scala> //Create the purposed view of tagged messages
    scala> val taggedMessageDS =
messageDataDS.filter(_.ShortMessage.contains("#")).map(message =>
(message.ShortMessage.split(" ").filter(_.contains("#")).mkString(" "),
message)).map(msgTuple => TaggedMessage(msgTuple._1, msgTuple._2.UserName,
msgTuple._2.MessageId, msgTuple._2.ShortMessage, msgTuple._2.Timestamp))
      taggedMessageDS: org.apache.spark.sql.Dataset[TaggedMessage] =
[HashTag: string, UserName: string ... 3 more fields]
    scala> //Convert the Dataset into data frame
    scala> val taggedMessageDF = taggedMessageDS.toDF()
      taggedMessageDF: org.apache.spark.sql.DataFrame = [HashTag: string,
UserName: string ... 3 more fields]
    scala> taggedMessageDF.createOrReplaceTempView("taggedMessages")
    scala> taggedMessageDF.show()
      +---------+--------+---------+--------------------+----------+
      |  HashTag|UserName|MessageId|        ShortMessage| Timestamp|
      +---------+--------+---------+--------------------+----------+
      |#Barcelona|  eharris|        8|Anybody knows goo...|1459011426|
      |#Barcelona|    wbrad|       12|#Barcelona has pl...|1459157132|
      +---------+--------+---------+--------------------+----------+
    scala> //The following are the queries given in the use cases
    scala> //Find the messages that are grouped by a given hash tag
    scala> val byHashTag = spark.sql("SELECT a.UserName, b.FirstName,
b.LastName, a.MessageId, a.ShortMessage, a.Timestamp FROM taggedMessages a,
user b WHERE a.UserName = b.UserName AND HashTag = '#Barcelona' ORDER BY
a.Timestamp")
      byHashTag: org.apache.spark.sql.DataFrame = [UserName: string,
FirstName: string ... 4 more fields]
    scala> byHashTag.show()
      +--------+---------+--------+---------+--------------------+---------
--+
      |UserName|FirstName|LastName|MessageId|        ShortMessage|
Timestamp|
      +--------+---------+--------+---------+--------------------+---------
```

```
-+
      | eharris|        Ed|  Harris|       8|Anybody knows
goo...|1459011426|
      |    wbrad|  William|Bradford|      12|#Barcelona has
pl...|1459157132|
      +--------+---------+--------+---------+-------------------+---------
-+
    scala> //Find the messages that are addressed to a given user
    scala> val byToUser = spark.sql("SELECT FromUserName, ToUserName,
MessageId, ShortMessage, Timestamp FROM messageToUsers WHERE ToUserName =
'wbryson' ORDER BY Timestamp")
      byToUser: org.apache.spark.sql.DataFrame = [FromUserName: string,
ToUserName: string ... 3 more fields]
    scala> byToUser.show()
      +------------+----------+---------+--------------------+----------+
      |FromUserName|ToUserName|MessageId|        ShortMessage| Timestamp|
      +------------+----------+---------+--------------------+----------+
      |       wbrad|   wbryson|        7|@wbryson Stuff ha...|1459011333|
      |      mtwain|   wbryson|       13|@wbryson It is go...|1459164906|
      +------------+----------+---------+--------------------+----------+
    scala> //Find the followers of a given user
    scala> val followers = spark.sql("SELECT b.FirstName as
FollowerFirstName, b.LastName as FollowerLastName, a.Followed FROM follow
a, user b WHERE a.Follower = b.UserName AND a.Followed = 'wbryson'")
      followers: org.apache.spark.sql.DataFrame = [FollowerFirstName:
string, FollowerLastName: string ... 1 more field]
    scala> followers.show()
      +-----------------+----------------+--------+
      |FollowerFirstName|FollowerLastName|Followed|
      +-----------------+----------------+--------+
      |          William|        Bradford| wbryson|
      |           Thomas|           Hardy| wbryson|
      +-----------------+----------------+--------+
    scala> //Find the followedUsers of a given user
    scala> val followedUsers = spark.sql("SELECT b.FirstName as
FollowedFirstName, b.LastName as FollowedLastName, a.Follower FROM follow
a, user b WHERE a.Followed = b.UserName AND a.Follower = 'eharris'")
      followedUsers: org.apache.spark.sql.DataFrame = [FollowedFirstName:
string, FollowedLastName: string ... 1 more field]
    scala> followedUsers.show()
      +-----------------+----------------+--------+
      |FollowedFirstName|FollowedLastName|Follower|
      +-----------------+----------------+--------+
      |           Thomas|            Cook| eharris|
      |             Mark|          Thomas| eharris|
      +-----------------+----------------+--------+
```

In the preceding Scala code snippet, the dataset and DataFrame-based programming model is being used because the programming language of choice was Scala. Now, since Python is not a strongly typed language, the Dataset API is not supported in Python, hence the following Python code uses the traditional RDD-based programming model of Spark in conjunction with the DataFrame-based programming model. At the Python REPL prompt, try the following statements:

```
>>> from pyspark.sql import Row
>>> #TODO: Change the following directory to point to your data
directory
>>> dataDir = "/Users/RajT/Documents/Writing/SparkForBeginners/To-
PACKTPUB/Contents/B05289-09-DesigningSparkApplications/Code/Data/"
>>> #Load the user data into an RDD
>>> userDataRDD = sc.textFile(dataDir + "user.txt").map(lambda line:
line.split("|")).map(lambda p: Row(Id=int(p[0]), UserName=p[1],
FirstName=p[2], LastName=p[3], EMail=p[4], AlternateEmail=p[5],
Phone=p[6]))
>>> #Convert the RDD into data frame
>>> userDataDF = userDataRDD.toDF()
>>> userDataDF.createOrReplaceTempView("user")
>>> userDataDF.show()
    +---------------+-------------------+---------+---+--------+-------
-------+---------+
    |  AlternateEmail|              EMail|FirstName| Id|LastName|
Phone|UserName|
    +---------------+-------------------+---------+---+--------+-------
-------+---------+
    |mt12@example.com|  mthomas@example.com|     Mark|  1|
Thomas|+4411860297701|  mthomas|
    |  mit@example.com|mithomas@example.com|  Michael|  2|
Thomas|+4411860297702|mithomas|
    |  mtw@example.com|   mtwain@example.com|     Mark|  3|
Twain|+4411860297703|   mtwain|
    |   th@example.com|   thardy@example.com|   Thomas|  4|
Hardy|+4411860297704|   thardy|
    |   bb@example.com| wbryson@example.com|  William|  5|
Bryson|+4411860297705| wbryson|
    |   wb@example.com|   wbrad@example.com|  William|
6|Bradford|+4411860297706|    wbrad|
    |   eh@example.com| eharris@example.com|       Ed|  7|
Harris|+4411860297707| eharris|
    |   tk@example.com|   tcook@example.com|   Thomas|  8|
Cook|+4411860297708|    tcook|
    |   ar@example.com| arobert@example.com|     Adam|  9|
Robert|+4411860297709| arobert|
    |   jj@example.com| jjames@example.com|    Jacob| 10|
James|+4411860297710|   jjames|
```

```
   +---------------+-------------------+--------+---+--------+------
-------+--------+
   >>> #Load the follower data into an RDD
   >>> followerDataRDD = sc.textFile(dataDir + "follower.txt").map(lambda
line: line.split("|")).map(lambda p: Row(Follower=p[0], Followed=p[1]))
   >>> #Convert the RDD into data frame
   >>> followerDataDF = followerDataRDD.toDF()
   >>> followerDataDF.createOrReplaceTempView("follow")
   >>> followerDataDF.show()
    +--------+--------+
    |Followed|Follower|
    +--------+--------+
    |mithomas| mthomas|
    |  mtwain| mthomas|
    | wbryson|  thardy|
    | wbryson|   wbrad|
    | mthomas| eharris|
    |   tcook| eharris|
    |  jjames| arobert|
    +--------+--------+
   >>> #Load the message data into an RDD
   >>> messageDataRDD = sc.textFile(dataDir + "message.txt").map(lambda
line: line.split("|")).map(lambda p: Row(UserName=p[0],
MessageId=int(p[1]), ShortMessage=p[2], Timestamp=int(p[3])))
   >>> #Convert the RDD into data frame
   >>> messageDataDF = messageDataRDD.toDF()
   >>> messageDataDF.createOrReplaceTempView("message")
   >>> messageDataDF.show()
    +---------+--------------------+----------+--------+
    |MessageId|        ShortMessage| Timestamp|UserName|
    +---------+--------------------+----------+--------+
    |        1|@mithomas Your po...|1459009608| mthomas|
    |        2|Feeling awesome t...|1459010608| mthomas|
    |        3|My namesake in th...|1459010776|  mtwain|
    |        4|Started the day w...|1459011016|  mtwain|
    |        5|It is just spring...|1459011199|  thardy|
    |        6|Some days are rea...|1459011256| wbryson|
    |        7|@wbryson Stuff ha...|1459011333|   wbrad|
    |        8|Anybody knows goo...|1459011426| eharris|
    |        9|Stock market is p...|1459011483|   tcook|
    |       10|Dont do day tradi...|1459011539|   tcook|
    |       11|I have never hear...|1459011622|   tcook|
    |       12|#Barcelona has pl...|1459157132|   wbrad|
    |       13|@wbryson It is go...|1459164906|  mtwain|
    +---------+--------------------+----------+--------+
```

These steps complete the process of loading all the required data from persistent stores into DataFrames. Here, the data comes from text files. In real-world use cases, it may come from popular NoSQL data stores, traditional RDBMS tables, or from Avro or Parquet serialized data stores loaded from HDFS. The following section uses these DataFrames and creates various purposed views and queries:

```
>>> #Create the purposed view of the message to users
>>> messagetoUsersRDD = messageDataRDD.filter(lambda message: "@" in
message.ShortMessage).map(lambda message : (message, " ".join(filter(lambda
s: s[0] == '@', message.ShortMessage.split(" "))))).map(lambda msgTuple:
Row(FromUserName=msgTuple[0].UserName, ToUserName=msgTuple[1][1:],
MessageId=msgTuple[0].MessageId, ShortMessage=msgTuple[0].ShortMessage,
Timestamp=msgTuple[0].Timestamp))
>>> #Convert the RDD into data frame
>>> messagetoUsersDF = messagetoUsersRDD.toDF()
>>> messagetoUsersDF.createOrReplaceTempView("messageToUsers")
>>> messagetoUsersDF.show()
    +------------+---------+--------------------+---------+----------+
    |FromUserName|MessageId|        ShortMessage| Timestamp|ToUserName|
    +------------+---------+--------------------+---------+----------+
    |     mthomas|        1|@mithomas Your po...|1459009608|  mithomas|
    |       wbrad|        7|@wbryson Stuff ha...|1459011333|   wbryson|
    |      mtwain|       13|@wbryson It is go...|1459164906|   wbryson|
    +------------+---------+--------------------+---------+----------+
>>> #Create the purposed view of tagged messages
>>> taggedMessageRDD = messageDataRDD.filter(lambda message: "#" in
message.ShortMessage).map(lambda message : (message, " ".join(filter(lambda
s: s[0] == '#', message.ShortMessage.split(" "))))).map(lambda msgTuple:
Row(HashTag=msgTuple[1], UserName=msgTuple[0].UserName,
MessageId=msgTuple[0].MessageId, ShortMessage=msgTuple[0].ShortMessage,
Timestamp=msgTuple[0].Timestamp))
>>> #Convert the RDD into data frame
>>> taggedMessageDF = taggedMessageRDD.toDF()
>>> taggedMessageDF.createOrReplaceTempView("taggedMessages")
>>> taggedMessageDF.show()
    +----------+---------+--------------------+---------+--------+
    |   HashTag|MessageId|        ShortMessage| Timestamp|UserName|
    +----------+---------+--------------------+---------+--------+
    |#Barcelona|        8|Anybody knows goo...|1459011426| eharris|
    |#Barcelona|       12|#Barcelona has pl...|1459157132|   wbrad|
    +----------+---------+--------------------+---------+--------+
>>> #The following are the queries given in the use cases
>>> #Find the messages that are grouped by a given hash tag
>>> byHashTag = spark.sql("SELECT a.UserName, b.FirstName, b.LastName,
a.MessageId, a.ShortMessage, a.Timestamp FROM taggedMessages a, user b
WHERE a.UserName = b.UserName AND HashTag = '#Barcelona' ORDER BY
a.Timestamp")
```

```
>>> byHashTag.show()
+-------+---------+--------+---------+-------------------+---------
-+
|UserName|FirstName|LastName|MessageId|       ShortMessage|
Timestamp|
+-------+---------+--------+---------+-------------------+---------
-+
| eharris|       Ed|  Harris|        8|Anybody knows
goo...|1459011426|
|   wbrad|  William|Bradford|       12|#Barcelona has
pl...|1459157132|
+-------+---------+--------+---------+-------------------+---------
-+
>>> #Find the messages that are addressed to a given user
>>> byToUser = spark.sql("SELECT FromUserName, ToUserName, MessageId,
ShortMessage, Timestamp FROM messageToUsers WHERE ToUserName = 'wbryson'
ORDER BY Timestamp")
>>> byToUser.show()
+------------+----------+---------+--------------------+----------+
|FromUserName|ToUserName|MessageId|        ShortMessage| Timestamp|
+------------+----------+---------+--------------------+----------+
|       wbrad|   wbryson|        7|@wbryson Stuff ha...|1459011333|
|      mtwain|   wbryson|       13|@wbryson It is go...|1459164906|
+------------+----------+---------+--------------------+----------+
>>> #Find the followers of a given user
>>> followers = spark.sql("SELECT b.FirstName as FollowerFirstName,
b.LastName as FollowerLastName, a.Followed FROM follow a, user b WHERE
a.Follower = b.UserName AND a.Followed = 'wbryson'")
>>> followers.show()
+-----------------+----------------+--------+
|FollowerFirstName|FollowerLastName|Followed|
+-----------------+----------------+--------+
|          William|        Bradford| wbryson|
|           Thomas|           Hardy| wbryson|
+-----------------+----------------+--------+
>>> #Find the followed users of a given user
>>> followedUsers = spark.sql("SELECT b.FirstName as FollowedFirstName,
b.LastName as FollowedLastName, a.Follower FROM follow a, user b WHERE
a.Followed = b.UserName AND a.Follower = 'eharris'")
>>> followedUsers.show()
+-----------------+----------------+--------+
|FollowedFirstName|FollowedLastName|Follower|
+-----------------+----------------+--------+
|           Thomas|            Cook| eharris|
|             Mark|          Thomas| eharris|
+-----------------+----------------+--------+
```

The purposed views and queries required to implement the use cases are developed as a single application. But in reality, it is not a good design practice to have all the views and queries in one application. It is good to separate them by persisting the views and refreshing them at regular intervals. If using only one application, caching and the use of custom-made context objects that are broadcasted to the Spark cluster could be employed to access the views.

Understanding custom data processes

The views created here were created to serve various queries and to produce desired outputs. There are some other classes of data processing applications that are often developed to implement real-world use cases. From the Lambda Architecture perspective, this also falls into the serving layer. The reason why these custom data processes fall into the serving layer is mainly because most of these use or process data from the master dataset and create views or outputs. It is also very possible for the custom processed data to remain as a view, and the following use case is one of such cases.

In the SfbMicroBlog microblogging application, it is a very common requirement to see whether a given user A is in some way connected to user B in a direct follower relationship or in a transitive way. This use case can be implemented using a graph data structure to see whether the two users in question are in the same connected component, whether they are connected in a transitive way, or whether they are not connected at all. For this, a graph is constructed with all the users as the vertices and the follow relationship as edges using a Spark GraphX library-based Spark application. At the Scala REPL prompt, try the following statements:

```scala
scala> import org.apache.spark.rdd.RDD
     import org.apache.spark.rdd.RDD
scala> import org.apache.spark.graphx._
     import org.apache.spark.graphx._
scala> //TODO: Change the following directory to point to your data
directory
scala> val dataDir = "/Users/RajT/Documents/Writing/SparkForBeginners/To-
PACKTPUB/Contents/B05289-09-DesigningSparkApplications/Code/Data/"
dataDir: String = /Users/RajT/Documents/Writing/SparkForBeginners/To-
PACKTPUB/Contents/B05289-09-DesigningSparkApplications/Code/Data/
scala> //Define the case classes in Scala for the entities
scala> case class User(Id: Long, UserName: String, FirstName: String,
LastName: String, EMail: String, AlternateEmail: String, Phone: String)
     defined class User
scala> case class Follow(Follower: String, Followed: String)
     defined class Follow
scala> case class ConnectedUser(CCId: Long, UserName: String)
```

```
        defined class ConnectedUser
scala> //Define the utility functions that are to be passed in the
applications
scala> def toUser =  (line: Seq[String]) => User(line(0).toLong, line(1),
line(2),line(3), line(4), line(5), line(6))
      toUser: Seq[String] => User
scala> def toFollow =  (line: Seq[String]) => Follow(line(0), line(1))
      toFollow: Seq[String] => Follow
scala> //Load the user data into an RDD
scala> val userDataRDD = sc.textFile(dataDir +
"user.txt").map(_.split("\\|")).map(toUser(_))
userDataRDD: org.apache.spark.rdd.RDD[User] = MapPartitionsRDD[160] at map
at <console>:34
scala> //Convert the RDD into data frame
scala> val userDataDF = userDataRDD.toDF()
userDataDF: org.apache.spark.sql.DataFrame = [Id: bigint, UserName: string
... 5 more fields]
scala> userDataDF.createOrReplaceTempView("user")
 scala> userDataDF.show()
      +---+--------+---------+--------+-----------+--------------+-------
-------+
Id|UserName|FirstName|LastName| EMail|  AlternateEmail|   Phone|
      +---+--------+---------+--------+-----------+--------------+-----------
---+
|  1| mthomas|     Mark|  Thomas| mthomas@example.com|mt12@example.com|
+4411860297701|
|  2|mithomas|  Michael|  Thomas|mithomas@example.com| mit@example.com|
+4411860297702|
|  3|  mtwain|     Mark|   Twain| mtwain@example.com| mtw@example.com|
+4411860297703|
|  4|  thardy|   Thomas|   Hardy| thardy@example.com|  th@example.com|
+4411860297704|
|  5| wbryson|  William|  Bryson| wbryson@example.com|  bb@example.com|
+4411860297705|
|  6|   wbrad|  William|Bradford|   wbrad@example.com|  wb@example.com|
+4411860297706|
|  7| eharris|       Ed|  Harris| eharris@example.com|  eh@example.com|
+4411860297707|
|  8|   tcook|   Thomas|    Cook|   tcook@example.com|  tk@example.com|
+4411860297708|
|  9| arobert|     Adam|  Robert| arobert@example.com|  ar@example.com|
+4411860297709|
| 10|  jjames|    Jacob|   James| jjames@example.com|  jj@example.com|
+4411860297710|
      +---+--------+---------+--------+-------------+--------------+-------
-------+
scala> //Load the follower data into an RDD
scala> val followerDataRDD = sc.textFile(dataDir +
```

```
  "follower.txt").map(_.split("\\|")).map(toFollow(_))
followerDataRDD: org.apache.spark.rdd.RDD[Follow] = MapPartitionsRDD[168]
at map at <console>:34
scala> //Convert the RDD into data frame
scala> val followerDataDF = followerDataRDD.toDF()
followerDataDF: org.apache.spark.sql.DataFrame = [Follower: string,
Followed: string]
scala> followerDataDF.createOrReplaceTempView("follow")
scala> followerDataDF.show()
      +--------+--------+
      |Follower|Followed|
      +--------+--------+
      | mthomas|mithomas|
      | mthomas|  mtwain|
      |  thardy| wbryson|
      |   wbrad| wbryson|
      | eharris| mthomas|
      | eharris|   tcook|
      | arobert|  jjames|
      +--------+--------+
scala> //By joining with the follower and followee users with the master
user data frame for extracting the unique ids
scala> val fullFollowerDetails = spark.sql("SELECT b.Id as FollowerId, c.Id
as FollowedId, a.Follower, a.Followed FROM follow a, user b, user c WHERE
a.Follower = b.UserName AND a.Followed = c.UserName")
fullFollowerDetails: org.apache.spark.sql.DataFrame = [FollowerId: bigint,
FollowedId: bigint ... 2 more fields]
scala> fullFollowerDetails.show()
      +----------+----------+--------+--------+
      |FollowerId|FollowedId|Follower|Followed|
      +----------+----------+--------+--------+
      |         9|        10| arobert|  jjames|
      |         1|         2| mthomas|mithomas|
      |         7|         8| eharris|   tcook|
      |         7|         1| eharris| mthomas|
      |         1|         3| mthomas|  mtwain|
      |         6|         5|   wbrad| wbryson|
      |         4|         5|  thardy| wbryson|
      +----------+----------+--------+--------+
scala> //Create the vertices of the connections graph
scala> val userVertices: RDD[(Long, String)] = userDataRDD.map(user =>
(user.Id, user.UserName))
userVertices: org.apache.spark.rdd.RDD[(Long, String)] =
MapPartitionsRDD[194] at map at <console>:36
scala> userVertices.foreach(println)
      (6,wbrad)
      (7,eharris)
      (8,tcook)
```

```
        (9,arobert)
        (10,jjames)
        (1,mthomas)
        (2,mithomas)
        (3,mtwain)
        (4,thardy)
        (5,wbryson)
scala> //Create the edges of the connections graph
scala> val connections: RDD[Edge[String]] =
fullFollowerDetails.rdd.map(conn => Edge(conn.getAs[Long]("FollowerId"),
conn.getAs[Long]("FollowedId"), "Follows"))
    connections:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[String]] =
MapPartitionsRDD[217] at map at <console>:29
    scala> connections.foreach(println)
    Edge(9,10,Follows)
    Edge(7,8,Follows)
    Edge(1,2,Follows)
    Edge(7,1,Follows)
    Edge(1,3,Follows)
    Edge(6,5,Follows)
    Edge(4,5,Follows)
    scala> //Create the graph using the vertices and the edges
    scala> val connectionGraph = Graph(userVertices, connections)
      connectionGraph: org.apache.spark.graphx.Graph[String,String] =
org.apache.spark.graphx.impl.GraphImpl@3c207acd
```

The user graph with the users in the vertices and the connection relationship forming the edges is done. On this graph data structure, run the graph processing algorithm, the connected component algorithm. The following code snippet does this:

```
    scala> //Calculate the connected users
    scala> val cc = connectionGraph.connectedComponents()
      cc:
org.apache.spark.graphx.Graph[org.apache.spark.graphx.VertexId,String] =
org.apache.spark.graphx.impl.GraphImpl@73f0bd11
    scala> // Extract the triplets of the connected users
    scala> val ccTriplets = cc.triplets
      ccTriplets:
org.apache.spark.rdd.RDD[org.apache.spark.graphx.EdgeTriplet[org.apache.spa
rk.graphx.VertexId,String]] = MapPartitionsRDD[285] at mapPartitions at
GraphImpl.scala:48
    scala> // Print the structure of the triplets
    scala> ccTriplets.foreach(println)
      ((9,9),(10,9),Follows)
      ((1,1),(2,1),Follows)
      ((7,1),(8,1),Follows)
      ((7,1),(1,1),Follows)
```

```
       ((1,1),(3,1),Follows)
       ((4,4),(5,4),Follows)
       ((6,4),(5,4),Follows)
```

The connected component graph, `cc`, and its triplets, `ccTriplets`, are created, and this can now be used to run various queries. Since the graph is an RDD-based data structure, if it is necessary to do queries, converting the graph RDD to DataFrames is a common practice. The following code demonstrates this:

```
     scala> //Print the vertex numbers and the corresponding connected
component id. The connected component id is generated by the system and it
is to be taken only as a unique identifier for the connected component
     scala> val ccProperties = ccTriplets.map(triplet => "Vertex " +
triplet.srcId + " and " + triplet.dstId + " are part of the CC with id " +
triplet.srcAttr)
     ccProperties: org.apache.spark.rdd.RDD[String] =
MapPartitionsRDD[288] at map at <console>:48
     scala> ccProperties.foreach(println)
       Vertex 9 and 10 are part of the CC with id 9
       Vertex 1 and 2 are part of the CC with id 1
       Vertex 7 and 8 are part of the CC with id 1
       Vertex 7 and 1 are part of the CC with id 1
       Vertex 1 and 3 are part of the CC with id 1
       Vertex 4 and 5 are part of the CC with id 4
       Vertex 6 and 5 are part of the CC with id 4
     scala> //Find the users in the source vertex with their CC id
     scala> val srcUsersAndTheirCC = ccTriplets.map(triplet =>
(triplet.srcId, triplet.srcAttr))
       srcUsersAndTheirCC:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
org.apache.spark.graphx.VertexId)] = MapPartitionsRDD[289] at map at
<console>:48
     scala> //Find the users in the destination vertex with their CC id
     scala> val dstUsersAndTheirCC = ccTriplets.map(triplet =>
(triplet.dstId, triplet.dstAttr))
       dstUsersAndTheirCC:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
org.apache.spark.graphx.VertexId)] = MapPartitionsRDD[290] at map at
<console>:48
     scala> //Find the union
     scala> val usersAndTheirCC =
srcUsersAndTheirCC.union(dstUsersAndTheirCC)
       usersAndTheirCC:
org.apache.spark.rdd.RDD[(org.apache.spark.graphx.VertexId,
org.apache.spark.graphx.VertexId)] = UnionRDD[291] at union at <console>:52
     scala> //Join with the name of the users
     scala> //Convert the RDD to DataFrame
     scala> val usersAndTheirCCWithName =
```

```
usersAndTheirCC.join(userVertices).map{case (userId,(ccId,userName)) =>
(ccId, userName)}.distinct.sortByKey().map{case (ccId,userName) =>
ConnectedUser(ccId, userName)}.toDF()
      usersAndTheirCCWithName: org.apache.spark.sql.DataFrame = [CCId:
bigint, UserName: string]
    scala> usersAndTheirCCWithName.createOrReplaceTempView("connecteduser")
    scala> val usersAndTheirCCWithDetails = spark.sql("SELECT a.CCId,
a.UserName, b.FirstName, b.LastName FROM connecteduser a, user b WHERE
a.UserName = b.UserName ORDER BY CCId")
      usersAndTheirCCWithDetails: org.apache.spark.sql.DataFrame = [CCId:
bigint, UserName: string ... 2 more fields]
    scala> //Print the usernames with their CC component id. If two users
share the same CC id, then they are connected
    scala> usersAndTheirCCWithDetails.show()
      +----+--------+---------+--------+
      |CCId|UserName|FirstName|LastName|
      +----+--------+---------+--------+
      |   1|mithomas|  Michael|  Thomas|
      |   1|  mtwain|     Mark|   Twain|
      |   1|   tcook|   Thomas|    Cook|
      |   1| eharris|       Ed|  Harris|
      |   1| mthomas|     Mark|  Thomas|
      |   4|   wbrad|  William|Bradford|
      |   4| wbryson|  William|  Bryson|
      |   4|  thardy|   Thomas|   Hardy|
      |   9|  jjames|    Jacob|   James|
      |   9| arobert|     Adam|  Robert|
      +----+--------+---------+--------+
```

Using the preceding implementation of a purposed view to get a list of users and their connected component identification numbers, if there is a need to find out whether two users are connected, just read the records of those two users and see whether they have the same connected component identification number.

References

For more information, visit the following links:

- http://lambda-architecture.net/
- https://www.dre.vanderbilt.edu/~schmidt/PDF/Context-Object-Pattern.pdf

Summary

This chapter concludes the book with one single application's use cases, implemented using the Spark concepts learned in the earlier chapters of the book. From a data processing application architecture perspective, this chapter covered the Lambda Architecture as a technology-agnostic architectural framework for data processing applications, which has huge applicability in the big data application development space.

From a data processing application development perspective, RDD-based Spark programming, Dataset-based Spark programming, Spark SQL-based DataFrames to process structured data, the Spark Streaming-based listener program that constantly listens to the incoming messages and processes them, and the Spark GraphX-based application to process follower relationships have been covered. The use cases covered so far have immense scope for readers to add their own functionalities and enhance the application use cases discussed in this chapter.

Index