This book will teach you the techniques and recipes for large-scale graph processing using Apache Spark. It is a step-by-step and detailed guide that will prove essential to anyone with an interest in, and need to process, large graphs.

This book is a step-by-step guide where each chapter teaches important techniques for different stages of the pipeline, from loading and transforming graph data to implementing graph-parallel operations and machine learning algorithms. It also uses a hands-on approach. We show how each technique works, using the Scala REPL with simple examples, and build standalone Spark applications. The book has detailed, complete, and tested Scala codes.

The book is loaded with real-world examples, and it applies the techniques presented to open datasets collected from various applications, ranging from social networks to food science and sports analytics.

Who this book is written for
This book is written for data scientists and big data developers who want to quickly learn how to perform graph processing with Apache Spark. It is an essential guide for anyone with a need to process or analyze large graphs.

What you will learn from this book
- Write, build, and deploy Spark applications with the Scala Build Tool
- Build and analyze large-scale network datasets
- Analyze and transform graphs using RDD and graph-specific operations
- Implement new custom graph operations tailored to specific needs
- Develop iterative and efficient graph algorithms using message aggregation and Pregel abstraction
- Extract subgraphs and use them to discover common clusters
- Analyze graph data and solve various data science problems using real-world datasets


Community Experience Distilled
Apache Spark Graph Processing
Build, process, and analyze large-scale graphs with Spark
Foreword by Denny Lee, Technology Evangelist, Databricks
Advisor, WearHacks

Rindra Ramamonjison

$ 34.99 US
£ 22.99 UK

Prices do not include local sales tax or VAT where applicable
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with Spark and GraphX'
- A synopsis of the book’s content
- More information on Apache Spark Graph Processing
Rindra Ramamonjison is a fourth year PhD student of electrical engineering at the University of British Columbia, Vancouver. He received his master’s degree from Tokyo Institute of Technology. He has played various roles in many engineering companies, within telecom and finance industries. His primary research interests are machine learning, optimization, graph processing, and statistical signal processing. Rindra is also the co-organizer of the Vancouver Spark Meetup.
Preface

This book is intended to present the GraphX library for Apache Spark and to teach the fundamental techniques and recipes to process graph data at scale. It is intended to be a self-study step-by-step guide for anyone new to Spark with an interest in or need for large-scale graph processing.

Distinctive features

The focus of this book is on large-scale graph processing with Apache Spark. The book teaches a variety of graph processing abstractions and algorithms and provides concise and sufficient information about them. You can confidently learn all of it and put it to use in different applications.

- **Step-by-step guide**: Each chapter teaches important techniques for every stage of the pipeline, from loading and transforming graph data to implementing graph-parallel operations and machine learning algorithms.
- **Hands-on approach**: We show how each technique works using the Scala REPL with simple examples and by building standalone Spark applications.
- **Detailed code**: All the Scala code in the book is available for download from the book webpage of Packt Publishing.
- **Real-world examples**: We apply these techniques on open datasets collected from a broad variety of applications ranging from social networks to food science and sports analytics.
What this book covers

This book consists of seven chapters. The first three chapters help you to get started quickly with Spark and GraphX. Then, the next two chapters teach the core techniques and abstractions to manipulate and aggregate graph data. Finally, the last two chapters of this book cover more advanced topics such as graph clustering, implementing graph-parallel iterative algorithms with Pregel, and learning methods from graph data.

Chapter 1, Getting Started with Spark and GraphX, begins with an introduction to the Spark system, its libraries, and the Scala Build Tool. It explains how to install and leverage Spark on the command line and in a standalone Scala program.

Chapter 2, Building and Exploring Graphs, presents the methods for building Spark graphs using illustrative network datasets.

Chapter 3, Graph Analysis and Visualization, walks you through the process of exploring, visualizing, and analyzing different network characteristics.

Chapter 4, Transforming and Shaping Up Graphs to Your Needs, teaches you how to transform raw datasets into a usable form that is appropriate for later analysis.

Chapter 5, Creating Custom Graph Aggregation Operators, teaches you how to create custom graph operations that are tailored to your specific needs with efficiency in mind, using the powerful message-passing aggregation operator in Spark.

Chapter 6, Iterative Graph-Parallel Processing with Pregel, explains the inner workings of the Pregel computational model and describes some use cases.

Chapter 7, Learning Graph Structures, introduces graph clustering, which is useful for detecting communities in graphs and applies it to a social music database.
Getting Started with Spark and GraphX

Apache Spark is a cluster-computing platform for the processing of large distributed datasets. Data processing in Spark is both fast and easy, thanks to its optimized parallel computation engine and its flexible and unified API. The core abstraction in Spark is based on the concept of Resilient Distributed Dataset (RDD). By extending the MapReduce framework, Spark’s Core API makes analytics jobs easier to write. On top of the Core API, Spark offers an integrated set of high-level libraries that can be used for specialized tasks such as graph processing or machine learning. In particular, GraphX is the library to perform graph-parallel processing in Spark.

This chapter will introduce you to Spark and GraphX by building a social network and exploring the links between people in the network. In addition, you will learn to use the Scala Build Tool (SBT) to build and run a Spark program. By the end of this chapter, you will know how to:

- Install Spark successfully on your computer
- Experiment with the Spark shell and review Spark’s data abstractions
- Create a graph and explore the links using base RDD and graph operations
- Build and submit a standalone Spark application with SBT

Downloading and installing Spark 1.4.1

In the following section, we will go through the Spark installation process in detail. Spark is built on Scala and runs on the Java Virtual Machine (JVM). Before installing Spark, you should first have Java Development Kit 7 (JDK) installed on your computer.

Next, download the latest release of Spark from the project website https://spark.apache.org/downloads.html. Perform the following three steps to get Spark installed on your computer:

1. Select the package type: Pre-built for Hadoop 2.6 and later and then Direct Download. Make sure you choose a prebuilt version for Hadoop instead of the source code.

2. Download the compressed TAR file called spark-1.4.1-bin-hadoop2.6.tgz and place it into a directory on your computer.

3. Open the terminal and change to the previous directory. Using the following commands, extract the TAR file, rename the Spark root folder to spark-1.4.1, and then list the installed files and subdirectories:

   ```
   tar -xf spark-1.4.1-bin-hadoop2.6.tgz
   mv spark-1.4.1-bin-hadoop2.6 spark-1.4.1
   cd spark-1.4.1
   ls
   ```

That's it! You now have Spark and its libraries installed on your computer. Note the following files and directories in the spark-1.4.1 home folder:

- **core**: This directory contains the source code for the core components and API of Spark
- **bin**: This directory contains the executable files that are used to submit and deploy Spark applications or also to interact with Spark in a Spark shell
- **graphx, mllib, sql, and streaming**: These are Spark libraries that provide a unified interface to do different types of data processing, namely graph processing, machine learning, queries, and stream processing
- **examples**: This directory contains demos and examples of Spark applications

It is often convenient to create shortcuts to the Spark home folder and Spark example folders. In Linux or Mac, open or create the ~/.bash_profile file in your home folder and insert the following lines:

```bash
export SPARKHOME="/[Where you put Spark]/spark-1.4.1/
export SPARKSCALAEX="ls ../spark-1.4.1/examples/src/main/scala/org/apache/spark/examples/"
```
Then, execute the following command for the previous shortcuts to take effect:

```
source ~/.bash_profile
```

As a result, you can quickly access these folders in the terminal or Spark shell. For example, the example named `LiveJournalPageRank.scala` can be accessed with:

```
$SPARKSCALAEX/graphx/LiveJournalPageRank.scala
```

### Experimenting with the Spark shell

The best way to learn Spark is through the Spark shell. There are two different shells for Scala and Python. But since the GraphX library is the most complete in Scala at the time this book was written, we are going to use the `spark-shell`, that is, the Scala shell. Let's launch the Spark shell inside the `$SPARKHOME/bin` from the command line:

```
$SPARKHOME/bin/spark-shell
```

If you set the current directory (cd) to `$SPARKHOME`, you can simply launch the shell with:

```
cd $SPARKHOME
./bin/spark-shell
```

If you happen to get an error saying something like: Failed to find Spark assembly in spark-1.4.1/assembly/target/scala-2.10. You need to build Spark before running this program, then it means that you have downloaded the Spark source code instead of a prebuilt version of Spark. In that case, go back to the project website and choose a prebuilt version of Spark.

If you were successful in launching the Spark shell, you should see the welcome message like this:

```
Welcome to

/ __/__  ___ _____/ /__
/_ \ _ \ / _ '/ __/  '_/
/___/ .__/
/___/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java)
```
For a sanity check, you can type in some Scala expressions or declarations and have them evaluated. Let's type some commands into the shell now:

```
scala> sc
res1: org.apache.spark.SparkContext = org.apache.spark.SparkContext@52e52233
scala> val myRDD = sc.parallelize(List(1,2,3,4,5))
myRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12
scala> sc.textFile("README.md").filter(line => line contains "Spark").count()
res2: Long = 21
```

Here is what you can tell about the preceding code. First, we displayed the Spark context defined by the variable `sc`, which is automatically created when you launch the Spark shell. The Spark context is the point of entry to the Spark API. Second, we created an RDD named `myRDD` that was obtained by calling the `parallelize` function for a list of five numbers. Finally, we loaded the `README.md` file into an RDD, filtered the lines that contain the word "Spark", and finally invoked an action on the filtered RDD to count the number of those lines.

It is expected that you are already familiar with the basic RDD transformations and actions, such as map, reduce, and filter. If that is not the case, I recommend that you learn them first, perhaps by reading the programming guide at https://spark.apache.org/docs/latest/programming-guide.html or an introductory book such as *Fast Data Processing with Spark* by Packt Publishing and *Learning Spark* by O'Reilly Media.

Don't panic if you did not fully grasp the mechanisms behind RDDs. The following refresher, however, helps you to remember the important points. RDD is the core data abstraction in Spark to represent a distributed collection of large datasets that can be partitioned and processed in parallel across a cluster of machines. The Spark API provides a uniform set of operations to transform and reduce the data within an RDD. On top of these abstractions and operations, the GraphX library also offers a flexible API that enables us to create graphs and operate on them easily.

Perhaps, when you ran the preceding commands in the Spark shell, you were overwhelmed by the long list of logging statements that start with `INFO`. There is a way to reduce the amount of information that Spark outputs in the shell.
You can reduce the level of verbosity of the Spark shell as follows:

• First, go to the $SCALAHOME/conf folder
• Then, create a new file called log4j.properties
• Inside the conf folder, open the template file log4j.properties.template and copy all its content into log4j.properties
• Find and replace the line log4j.rootCategory=INFO, console with either one of these two lines:
  • log4j.rootCategory=WARN, console
  • log4j.rootCategory=ERROR, console
• Finally, restart the Spark shell and you should now see fewer logging messages in the shell outputs

Getting started with GraphX

Now that we have installed Spark and experimented with the Spark shell, let's create our first graph in Spark by writing our code in the shell, and then building upon that code to develop and run a standalone program. We have three learning goals in this section:

1. First, you will learn how to construct and explore graphs using the Spark Core and GraphX API through a concrete example.
2. Second, you will review some important Scala programming features that are important to know when doing graph processing in Spark.
3. Third, you will learn how to develop and run a standalone Spark application.

Building a tiny social network

Let's create a tiny social network and explore the relationships among the different people in the network. Again, the best way to learn Spark is inside the shell. Our workflow is therefore to first experiment in the shell and then migrate our code later into a standalone Spark application. Before launching the shell, make sure to change the current directory to $SPARKHOME.

First, we need to import the GraphX and RDD module, as shown, so that we can invoke its APIs with their shorter names:

```
scala> import org.apache.spark.graphx._
scala> import org.apache.spark.rdd.RDD
```
As said previously, SparkContext is the main point of entry into a Spark program and it is created automatically in the Spark shell. It also offers useful methods to create RDDs from local collections, to load data from a local or Hadoop file system into RDDs, and to save output data on disks.

Loading the data

In this example, we will work with two CSV files people.csv and links.csv, which are contained in the directory $SPARKHOME/data/. Let's type the following commands to load these files into Spark:

```scala
scala> val people = sc.textFile("./data/people.csv")
people: org.apache.spark.rdd.RDD[String] = ./data/people.csv
  MappedRDD[81] at textFile at <console>:33

scala> val links = sc.textFile("./data/links.csv")
links: org.apache.spark.rdd.RDD[String] = ./data/links.csv
  MappedRDD[83] at textFile at <console>:33
```

Loading the CSV files just gave us back two RDDs of strings. To create our graph, we need to parse these strings into two suitable collections of vertices and edges.

It is important that your current directory inside the shell is $SPARKHOME. Otherwise, you get an error later because Spark cannot find the files.

The property graph

Before going further, let's introduce some key definitions and graph abstractions. In Spark, a graph is represented by a property graph, which is defined in the Graph class as:

```scala
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED, VD]
}
```
This means that the Graph class provides getters to access its vertices and its edges. These are later abstracted by the RDD subclasses VertexRDD[VD] and EdgeRDD[ED, VD]. Note that VD and ED here denote some Scala-type parameters of the classes VertexRDD, EdgeRDD, and Graph. These types of parameters can be primitive types, such as String, or also user-defined classes, such as the Person class, in our example of a social graph. It is important to note that the property graph in Spark is a directed multigraph. It means that the graph is permitted to have multiple edges between any pair of vertices. Moreover, each edge is directed and defines a unidirectional relationship. This is easy to grasp, for instance, in a Twitter graph where a user can follow another one but the converse does not need to be true. To model bidirectional links, such as a Facebook friendship, we need to define two edges between the nodes, and these edges should point in opposite directions. Additional properties about the relationship can be stored as an attribute of the edge.

A property graph is a graph with user-defined objects attached to each vertex and edge. The classes of these objects describe the properties of the graph. This is done in practice by parameterizing the class Graph, VertexRDD, and EdgeRDD. Moreover, each edge of the graph defines a unidirectional relationship but multiple edges can exist between any pair of vertices.

### Transforming RDDs to VertexRDD and EdgeRDD

Going back to our example, let's construct the graph in three steps, as follows:

1. We define a case class Person, which has name and age as class parameters. Case classes are very useful when we need to do pattern matching on an object Person later on:
   ```scala
case class Person(name: String, age: Int)
```

2. Next, we are going to parse each line of the CSV texts inside people and links into new objects of type Person and Edge respectively, and collect the results in RDD[(VertexId, Person)] and RDD[Edge[String]]:
   ```scala
   val peopleRDD: RDD[(VertexId, Person)] = people map { line =>
   case row = line split ' ','
     (row(0).toInt, Person(row(1), row(2).toInt))
   `}
scala> type Connection = String
scala> val linksRDD: RDD[Edge[Connection]] = links map {line =>
val row = line split ','
Edge(row(0).toInt, row(1).toInt, row(2))
}

To paste or write code in multiple lines in the shell:
- Type the command :paste
- Paste or write the given code
- Evaluate the code by pressing the keys Cmd + D on Mac or Ctrl + D in Windows

VertexId is simply a type alias for Long as defined in GraphX. In addition, the Edge class is defined in org.apache.spark.graphx.Edge as:

class Edge(srcId: VertexId, dstId: VertexId, attr: ED)

The class parameters srcId and dstId are the vertex IDs of the source and destination, which are linked by the edge. In our social network example, the link between two people is unidirectional and its property is described in the attr of type Connection. Note that we defined Connection as a type alias for String. For clarity, it often helps to give a meaningful name to the type parameter of Edge.

3. Now, we can create our social graph and name it tinySocial using the factory method Graph(...):

    scala> val tinySocial: Graph[Person, Connection] =
        | Graph(peopleRDD, linksRDD)
    tinySocial: org.apache.spark.graphx.Graph[Person,Connection] =
        | org.apache.spark.graphx.impl.GraphImpl@128cd92a

There are two things to note about this constructor. I told you earlier that the member vertices and edges of the graph are instances of VertexRDD[VD] and EdgeRDD[ED, VD]. However, we passed RDD[(VertexId, Person)] and RDD[Edge[Connection]] into the above factory method Graph. How did that work? It worked because VertexRDD[VD] and EdgeRDD[ED, VD] are subclasses of RDD[(VertexId, Person)] and RDD[Edge[Connection]] respectively. In addition, VertexRDD[VD] adds the constraint that VertexId occurs only once. Basically, two people in our social network cannot have the same vertex ID. Furthermore, VertexRDD[VD] and EdgeRDD[ED, VD] provide several other operations to transform vertex and edge attributes. We will see more of these in later chapters.
Introducing graph operations

Finally, we are going to look at the vertices and edges in the network by accessing and collecting them:

```scala
scala> tinySocial.vertices.collect()
res: Array[(org.apache.spark.graphx.VertexId, Person)] =
Array((4,Person(Dave,25)), (6,Person(Faith,21)), (8,Person(Harvey,47)),
(2,Person(Bob,18)), (1,Person(Alice,20)), (3,Person(Charlie,30)),
(7,Person(George,34)), (9,Person(Ivy,21)), (5,Person(Eve,30)))
scala> tinySocial.edges.collect()
res: Array[org.apache.spark.graphx.Edge[Connection]] =
Array(Edge(1,2,friend), Edge(1,3,sister), Edge(2,4,brother),
Edge(3,2,boss), Edge(4,5,client), Edge(1,9,friend), Edge(6,7,cousin),
Edge(7,9,coworker), Edge(8,9,father))
```

We used the edges and vertices getters in the Graph class and used the RDD action collect to put the result into a local collection.

Now, suppose we want to print only the professional connections that are listed in the following profLinks list:

```scala
val profLinks: List[Connection] = List("Coworker", "Boss",
"Employee","Client", "Supplier")
```

A bad way to arrive at the desired result is to filter the edges corresponding to professional connections, then loop through the filtered edges, extract the corresponding vertices' names, and print the connections between the source and destination vertices. We can write that method in the following code:

```scala
val profNetwork =
tinySocial.edges.filter{ case Edge(_, _, link) =>
profLinks.contains(link)}
for {
  Edge(src, dst, link) <- profNetwork.collect()
  srcName = (peopleRDD.filter{case (id, person) => id == src}
  first)._2.name
  dstName = (peopleRDD.filter{case (id, person) => id == dst}
  first)._2.name
  } println(srcName + " is a " + link + " of " + dstName)
```

Charlie is a boss of Bob
Dave is a client of Eve
George is a coworker of Ivy
Getting Started with Spark and GraphX

There are two problems with the preceding code. First, it could be more concise and expressive. Second, it is not efficient due to the filtering operations inside the for loop.

Luckily, there is a better alternative. The GraphX library provides two different ways to view data: either as a graph or as tables of edges, vertices, and triplets. For each view, the library offers a rich set operations whose implementations are optimized for execution. That means that we can often process a graph using a predefined graph operation or algorithm, easily. For instance, we could simplify the previous code and make it more efficient, as follows:

```scala
tinySocial.subgraph(profLinks contains _.attr).
  triplets.foreach(t => println(t.srcAttr.name + " is a " +
    t.attr + " of " + t.dstAttr.name))
Charlie is a boss of Bob
Dave is a client of Eve
George is a coworker of Ivy
```

We simply used the `subgraph` operation to filter the professional links. Then, we used the `triplet view` to access the attributes of the edges and vertices simultaneously. In brief, the triplet operator returns an RDD of `EdgeTriplet[Person, Connection]`. Note that `EdgeTriplet` is simply an alias for the parameterized type of 3-tuple `((VertexId, Person), (VertexId, Person), Connection)` that contains all the information about the source node, the destination node, and the edge property.

**Building and submitting a standalone application**

Let's conclude this chapter by developing and running a standalone Spark application for our social network example.

**Writing and configuring a Spark program**

Satisfied with our experiment in the shell, let's now write our first Spark program. Open your favorite text editor and create a new file named `simpleGraph.scala` and put it in the folder `$SPARKHOME/exercises/chap1`. A template for a Spark program looks like the following code:

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx._
object SimpleGraphApp {
  def main(args: Array[String]) {

    // Configure the program
    val conf = new SparkConf()
      .setAppName("Tiny Social")
      .setMaster("local")
      .set("spark.driver.memory", "2G")
    val sc = new SparkContext(conf)

    // Load some data into RDDs
    val people = sc.textFile("./data/people.csv")
    val links = sc.textFile("./data/links.csv")

    // After that, we use the Spark API as in the shell
    // ...
  }
}

You can also see the entire code of our SimpleGraph.scala file in the example files, which you can download from the Packt website.

**Downloading the example code**

You can download the example code files you purchased, as well as the code for all Packt books you have previously purchased, 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.

Let's go over this code to understand what is required to create and configure a Spark standalone program in Scala.

As a Scala program, our Spark application should be constructed within a top-level Scala object, which must have a main function that has the signature:

```scala
def main(args: Array[String]): Unit
```

In other words, the main program accepts an array of strings as a parameter and returns nothing. In our example, the top-level object is SimpleGraphApp.
Getting Started with Spark and GraphX

At the beginning of `simpleGraph.scala`, we have put the following import statements:

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

The first two lines import the `SparkContext` class as well as some implicit conversions defined in its companion object. It is not very important to know what the implicit conversions are. Just make sure you import both `SparkContext` and `SparkContext._`.

When we worked in the Spark shell, `SparkContext` and `SparkContext._` were imported automatically for us.

The third line imports `SparkConf`, which is a wrapper class that contains the configuration settings of a Spark application, such as its application name, the memory size of each executor, and the address of the master or cluster manager.

Next, we have imported some RDD and GraphX-specific class constructors and operators with these lines:

```scala
import org.apache.spark.rdd.RDD
import org.apache.spark.graphx._
```

The underscore after `org.apache.spark.graphx` makes sure that all public APIs in GraphX get imported.

Within `main`, we had to first configure the Spark program. To do this, we created an object called `SparkConf` and set the application settings through a chain of setter methods on the `SparkConf` object. `SparkConf` provides specific setters for some common properties, such as the application name or master. Alternatively, a generic `set` method can be used to set multiple properties together by passing them as a sequence of key-value pairs. The most common configuration parameters are listed in the following table with their default values and usage. The extensive list can be found at https://spark.apache.org/docs/latest/configuration.html:

<table>
<thead>
<tr>
<th>Spark property name</th>
<th>Usage and default value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>spark.app.name</code></td>
<td>This is the name of your application. This will appear in the UI and in the log data.</td>
</tr>
<tr>
<td>Spark property name</td>
<td>Usage and default value</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>spark.master</td>
<td>This is the cluster manager to connect to, for example, <code>spark://host:port</code>, <code>mesos://host:port</code>, <code>yarn</code>, or <code>local</code>.</td>
</tr>
<tr>
<td>spark.executor.memory</td>
<td>This is the amount of memory to use per executor process, in the same format as JVM memory strings (for example, <code>512 M</code>, <code>2 G</code>). The default value is <code>1 G</code>.</td>
</tr>
<tr>
<td>spark.driver.memory</td>
<td>When you run Spark locally with <code>spark.master=localhost</code>, your executor becomes the driver and you need to set this parameter instead of <code>spark.executor.memory</code>. The default value is <code>512 M</code>.</td>
</tr>
<tr>
<td>spark.storage.memoryFraction</td>
<td>This is the fraction of Java heap to use for Spark's memory cache. The default is <code>0.6</code>.</td>
</tr>
<tr>
<td>spark.serializer</td>
<td>This is the class used to serialize objects to be sent over the network or to be cached in serialized form. This is the subclass of the default class <code>org.apache.spark.serializer.JavaSerializer</code>.</td>
</tr>
</tbody>
</table>

In our example, we initialized the program as follows:

```scala
val conf = new SparkConf()
   .setAppName("Tiny Social")
   .setMaster("local")
   .set("spark.driver.memory", "2G")
val sc = new SparkContext(conf)
```

Precisely, we set the name of our application to "Tiny Social" and the master to be the local machine on which we submit the application. In addition, the driver memory is set to `2 GB`. Should we have set the master to be a cluster instead of local, we can specify the memory per executor by setting `spark.executor.memory` instead of `spark.driver.memory`.

In principle, the driver and executor have different roles and, in general, they run on different processes except when we set the master to be local. The driver is the process that compiles our program into tasks, schedules these tasks to one of more executors, and maintains the physical location of every RDD. Each executor is responsible for executing the tasks, and storing and caching RDDs in memory.
Getting Started with Spark and GraphX

It is not mandatory to set the Spark application settings in the SparkConf object inside your program. Alternatively, when submitting our application, we could set these parameters as command-line options of the spark-submit tool. We will cover that part in detail in the following sections. In this case, we will just create our SparkContext object as:

```scala
val sc = new SparkContext(new SparkConf())
```

After configuring the program, the next task is to load the data that we want to process by calling utility methods such as `sc.textFile` on the SparkContext object `sc`:

```scala
val people = sc.textFile("./data/people.csv")
val links = sc.textFile("./data/links.csv")
```

Finally, the rest of the program consists of the same operations on RDDs and graphs that we have used in the shell.

To avoid confusion when passing a relative file path to I/O actions such as `sc.textFile()`, the convention used in this book is that the current directory of the command line is always set to the project root folder. For instance, if our Tiny Social app's root folder is `$SPARKHOME/exercises/chap1`, then Spark will look for the data to be loaded in `$SPARKHOME/exercises/chap1/data`. This assumes that we have put the files in that data folder.

### Building the program with the Scala Build Tool

After writing our entire program, we are going to build it using the Scala Build Tool (SBT). If you do not have SBT installed on your computer yet, you need to install it first. Detailed instructions on how to install SBT are available at [http://www.scala-sbt.org/0.13/tutorial/index.html](http://www.scala-sbt.org/0.13/tutorial/index.html) for most operating systems. While there are different ways to install SBT, I recommend using a package manager instead of the manual installation. After the installation, execute the following command to append the SBT installation folder to the `PATH` environment variable:

```bash
$ export PATH=$PATH:/usr/local/bin/sbtl
```

Once we have SBT properly installed, we can use it to build our application with all its dependencies inside a single JAR package file, also called *uber jar*. In fact, when running a Spark application on several worker machines, an error will occur if some machines do not have the right dependency JAR.
By packaging an uber jar with SBT, the application code and its dependencies are all distributed to the workers. Concretely, we need to create a build definition file in which we set the project settings. Moreover, we must specify the dependencies and the resolvers that help SBT find the packages that are needed by our program. A resolver indicates the name and location of the repository that has the required JAR file. Let’s create a file called build.sbt in the project root folder $SPARKHOME/exercises/chap1 and insert the following lines:

```scala
name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.4.1",
  "org.apache.spark" %% "spark-graphx" % "1.4.1"
)

resolvers += "Akka Repository" at "http://repo.akka.io/releases/"
```

By convention, the settings are defined by Scala expressions and they need to be delimited by blank lines. Earlier, we set the project name, its version number, the version of Scala, as well as the Spark library dependencies. To build the program, we then enter the command:

```
$ sbt package
```

This will create a JAR file inside $SPARKHOME/exercises/chap1/target/scala-2.10/simple-project_2.10-1.0.jar.

### Deploying and running with spark-submit

Finally, we are going to invoke the spark-submit script in $SPARKHOME/bin/ to run the program from the root directory $SPARKHOME/exercises/chap1 in the terminal:

```
$ ../../bin/spark-submit --class "SimpleGraphApp" \
./target/scala-2.10/simple-project_2.10-1.0.jar
```

Spark assembly has been built with Hive, including Datanucleus jars on classpath.

Charlie is a boss of Bob
Dave is a client of Eve
George is a coworker of Ivy
The required options for the spark-submit command are the Scala application object name and the JAR file that we previously built with SBT. In case we did not set the master setting when creating the SparkConf object, we also would have to specify the --master option in spark-submit.

You can list all the available options for the spark-submit script with the command:

```
spark-submit --help
```

More details about how to submit a Spark application to a remote cluster are available at http://spark.apache.org/docs/latest/submitting-applications.html.

**Summary**

In this chapter, we took a whirlwind tour of graph processing in Spark. Specifically, we installed the Java Development Kit, a prebuilt version of Spark and the SBT tool. Furthermore, you were introduced to graph abstraction and operations in Spark by creating a social network in the Spark shell and also in a standalone program.

In the next chapter, you will learn more about how to build and explore graphs in Spark.
Where to buy this book

You can buy Apache Spark Graph Processing from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.