Chapter 2. Downloading Spark and Getting Started

In this chapter we will walk through the process of downloading and running Spark in local mode on a single computer. This chapter was written for anybody who is new to Spark, including both data scientists and engineers.

Spark can be used from Python, Java, or Scala. To benefit from this book, you don't need to be an expert programmer, but we do assume that you are comfortable with the basic syntax of at least one of these languages. We will include examples in all languages wherever possible.

Spark itself is written in Scala, and runs on the Java Virtual Machine (JVM). To run Spark on either your laptop or a cluster, all you need is an installation of Java 6 or newer. If you wish to use the Python API you will also need a Python interpreter (version 2.6 or newer). Spark does not yet work with Python 3.

Downloading Spark

The first step to using Spark is to download and unpack it. Let's start by downloading a recent precompiled released version of Spark. Visit *http://spark.apache.org/downloads.html* (http://spark.apache.org/downloads.html), select the package type of "Pre-built for Hadoop 2.4 and later," and click "Direct Download." This will download a compressed TAR file, or *tarball*, called *spark-1.2.0-bin-hadoop2.4.tgz*.

TIP

Windows users may run into issues installing Spark into a directory with a space in the name. Instead, install Spark in a directory with no space (e.g., *C:\spark*).

You don't need to have Hadoop, but if you have an existing Hadoop cluster or HDFS installation, download the matching version. You can do so from *http://spark.apache.org/downloads.html* (http://spark.apache.org/downloads.html) by selecting a different package type, but they will have slightly different filenames. Building from source is also possible; you can find the latest source code on GitHub (http://github.com/apache/spark) or select the package type of "Source Code" when downloading.

TIP

Most Unix and Linux variants, including Mac OS X, come with a command-line tool called `tar` that can be used to unpack TAR files. If your operating system does not have the `tar` command installed, try searching the Internet for a free TAR extractor—for example, on Windows, you may wish to try 7-Zip.

Now that we have downloaded Spark, let's unpack it and take a look at what comes with the default Spark distribution. To do that, open a terminal, change to the directory where you downloaded Spark, and untar the file. This will create a new directory with the same name but without the final *.tgz* suffix. Change into that directory and see what's inside. You can use the following commands to accomplish all of that:

```
cd ~
tar -xf spark-1.2.0-bin-hadoop2.4.tgz
cd spark-1.2.0-bin-hadoop2.4
ls
```

In the line containing the `tar` command, the `x` flag tells `tar` we are extracting files, and the `f` flag specifies the name of the tarball. The `ls` command lists the contents of the Spark directory. Let's briefly consider the names and purposes of some of the more important files and directories you see here that come with Spark:

README.md

Contains short instructions for getting started with Spark.

tion. We will cover the various deployment modes in detail in Chapter 7.

Introduction to Spark’s Python and Scala Shells

Spark comes with interactive shells that enable ad hoc data analysis. Spark’s shells will feel familiar if you have used other shells such as those in R, Python, and Scala, or operating system shells like Bash or the Windows command prompt.

Unlike most other shells, however, which let you manipulate data using the disk and memory on a single machine, Spark’s shells allow you to interact with data that is distributed on disk or in memory across many machines, and Spark takes care of automatically distributing this processing.

Because Spark can load data into memory on the worker nodes, many distributed computations, even ones that process terabytes of data across dozens of machines, can run in a few seconds. This makes the sort of iterative, ad hoc, and exploratory analysis commonly done in shells a good fit for Spark. Spark provides both Python and Scala shells that have been augmented to support connecting to a cluster.

TIP

Most of this book includes code in all of Spark’s languages, but interactive shells are available only in Python and Scala. Because a shell is very useful for learning the API, we recommend using one of these languages for these examples even if you are a Java developer. The API is similar in every language.

The easiest way to demonstrate the power of Spark’s shells is to start using one of them for some simple data analysis. Let’s walk through the example from the Quick Start Guide (http://spark.apache.org/docs/latest/quick-start.html) in the official Spark documentation.

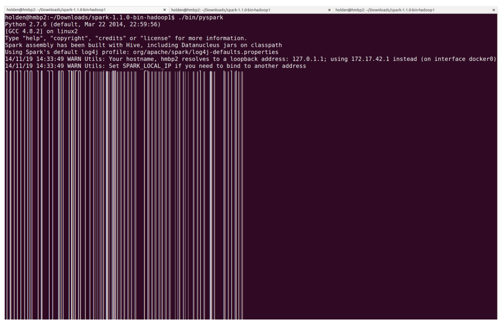
The first step is to open up one of Spark’s shells. To open the Python version of the Spark shell, which we also refer to as the PySpark Shell, go into your Spark directory and type:

```
bin/pyspark
```

(Or `bin\pyspark` in Windows.) To open the Scala version of the shell, type:

```
bin/spark-shell
```

The shell prompt should appear within a few seconds. When the shell starts, you will notice a lot of log messages. You may need to press Enter once to clear the log output and get to a shell prompt. Figure 2-1 shows what the PySpark shell looks like when you open it.



holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1
holden@hmbp2: ~/Downloads/spark-1.1.0-bin-hadoop1

```
holden@hmbp2:~/Downloads/spark-1.1.0-bin-hadoop1$ ./bin/pyspark
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/11/19 14:33:49 WARN Utils: Your hostname, hmbp2 resolves to a loopback address: 127.0.1.1; using 172.17.42.1 instead (on interface docker0)
14/11/19 14:33:49 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
```

Figure 2-1. The PySpark shell with default logging output

You may find the logging statements that get printed in the shell distracting. You can control the verbosity of the logging. To do this, you can create a file in the *conf* directory called *log4j.properties*. The Spark developers already include a template for this file called *log4j.properties.template*. To make the logging less verbose, make a copy of *conf/log4j.properties.template* called *conf/log4j.properties* and find the following line:

```
log4j.rootCategory=INFO, console
```

Figure 2-2. The PySpark shell with less logging output

USING IPYTHON

IPython is an enhanced Python shell that many Python users prefer, offering features such as tab completion. You can find instructions for installing it at *http://ipython.org* (http://ipython.org). You can use IPython with Spark by setting the `IPYTHON` environment variable to 1:

```
IPYTHON=1 ./bin/pyspark
```

To use the IPython Notebook, which is a web-browser-based version of IPython, use:

```
IPYTHON_OPTS="notebook" ./bin/pyspark
```

On Windows, set the variable and run the shell as follows:

```
set IPYTHON=1
bin\pyspark
```

In Spark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster. These collections are called *resilient distributed datasets*, or RDDs. RDDs are Spark's fundamental abstraction for distributed data and computation.

Before we say more about RDDs, let's create one in the shell from a local text file and do some very simple ad hoc analysis by following Example 2-1 for Python or Example 2-2 for Scala.

Example 2-1. Python line count

```
>>> lines = sc.textFile("README.md") # Create an RDD called line

>>> lines.count() # Count the number of items in this RDD
127
>>> lines.first() # First item in this RDD, i.e. first line of R
u'# Apache Spark'
```

Example 2-2. Scala line count

```
scala> val lines = sc.textFile("README.md") // Create an RDD ca
lines: spark.RDD[String] = MappedRDD[...]

scala> lines.count() // Count the number of items in this RDD
res0: Long = 127

scala> lines.first() // First item in this RDD, i.e. first line
res1: String = # Apache Spark
```

To exit either shell, press Ctrl-D.

TIP

We will discuss it more in Chapter 7, but one of the messages you may have noticed is `INFO SparkUI: Started SparkUI at http://[ipaddress]:4040`. You can access the Spark UI there and see all sorts of information about your tasks and cluster.

In Examples 2-1 and 2-2, the variable called `lines` is an RDD, created here from a text file on our local machine. We can run various parallel operations on the RDD, such as counting the number of elements in the dataset (here, lines of text in the file) or printing the first one. We will discuss RDDs in great depth in later chapters, but before we go any further, let's take a moment now to introduce basic Spark concepts.

Introduction to Core Spark Concepts

Now that you have run your first Spark code using the shell, it's time to learn about ferent machines might count lines in different ranges of the file. Because we just ran the Spark shell locally, it executed all its work on a single machine—but you can connect the same shell to a cluster to analyze data in parallel. Figure 2-3 shows how Spark executes on a cluster.

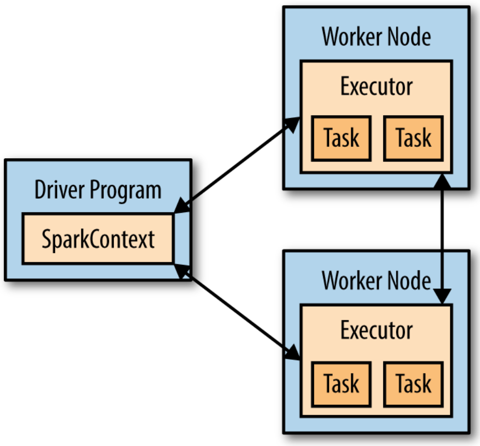


Figure 2-3. Components for distributed execution in Spark

Finally, a lot of Spark's API revolves around passing functions to its operators to run them on the cluster. For example, we could extend our README example by *filtering* the lines in the file that contain a word, such as *Python*, as shown in Example 2-4 (for Python) and Example 2-5 (for Scala).

Example 2-4. Python filtering example

```
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>> pythonLines.first()
u'## Interactive Python Shell'
```

Example 2-5. Scala filtering example

```
scala> val lines = sc.textFile("README.md") // Create an RDD ca
lines: spark.RDD[String] = MappedRDD[...]

scala> val pythonLines = lines.filter(line => line.contains("Py
pythonLines: spark.RDD[String] = FilteredRDD[...]

scala> pythonLines.first()
res0: String = ## Interactive Python Shell
```

Java 8 introduces shorthand syntax called *lambdas* that looks similar to Python and Scala. Here is how the code would look with this syntax:

```
JavaRDD<String> pythonLines = lines.filter(line -> line.c
```

We discuss passing functions further in "Passing Functions to Spark".

While we will cover the Spark API in more detail later, a lot of its magic is that function-based operations like `filter` *also* parallelize across the cluster. That is, Spark automatically takes your function (e.g., `line.contains("Python")`) and ships it to executor nodes. Thus, you can write code in a single driver program and automatically have parts of it run on multiple nodes. Chapter 3 covers the RDD API in detail.

Standalone Applications

The final piece missing in this quick tour of Spark is how to use it in standalone programs. Apart from running interactively, Spark can be linked into standalone applications in either Java, Scala, or Python. The main difference from using it in the shell is that you need to initialize your own SparkContext. After that, the API is the same.

The process of linking to Spark varies by language. In Java and Scala, you give your application a Maven dependency on the `spark-core` artifact. As of the time of writing, the latest Spark version is 1.2.0, and the Maven coordinates for that are:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.2.0
```

Maven is a popular package management tool for Java-based languages that lets you link to libraries in public repositories. You can use Maven itself to build your project, or use other tools that can talk to the Maven repositories, including Scala's sbt tool or Gradle. Popular integrated development environments like Eclipse also allow you to directly add a Maven dependency to a project.

In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark dependencies for us in Python. This script sets up the environment for Spark's Python API to function. Simply run your script with the line given in Example 2-6.

Example 2-6. Running a Python script

```
bin/spark-submit my_script.py
```

(Note that you will have to use backslashes instead of forward slashes on Windows.)

Initializing a SparkContext

Once you have linked an application to Spark, you need to import the Spark packages in your program and create a SparkContext. You do so by first creating a `SparkConf` object to configure your application, and then building a SparkContext for it. Examples 2-7 through 2-9 demonstrate this in each supported language.

Example 2-7. Initializing Spark in Python

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

Example 2-8. Initializing Spark in Scala

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

val conf = new SparkConf().setMaster("local").setAppName("My
val sc = new SparkContext(conf)
```

Example 2-9. Initializing Spark in Java

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName(
JavaSparkContext sc = new JavaSparkContext(conf);
```

code is automatically shipped to worker nodes. For now, please refer to the Quick Start Guide (http://bit.ly/1upkhfx) in the official Spark documentation.

Building Standalone Applications

This wouldn't be a complete introductory chapter of a Big Data book if we didn't have a word count example. On a single machine, implementing word count is simple, but in distributed frameworks it is a common example because it involves reading and combining data from many worker nodes. We will look at building and packaging a simple word count example with both sbt and Maven. All of our examples can be built together, but to illustrate a stripped-down build with minimal dependencies we have a separate smaller project underneath the *learning-spark-examples/mini-complete-example* directory, as you can see in Examples 2-10 (Java) and 2-11 (Scala).

Example 2-10. Word count Java application—don't worry about the details yet

```
// Create a Java Spark Context
SparkConf conf = new SparkConf().setAppName("wordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
// Load our input data.
JavaRDD<String> input = sc.textFile(inputFile);
// Split up into words.
JavaRDD<String> words = input.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String x) {
      return Arrays.asList(x.split(" "));
    }});
// Transform into pairs and count.
JavaPairRDD<String, Integer> counts = words.mapToPair(
  new PairFunction<String, String, Integer>(){
    public Tuple2<String, Integer> call(String x){
      return new Tuple2(x, 1);
    }}).reduceByKey(new Function2<Integer, Integer, Integer>(){
        public Integer call(Integer x, Integer y){ return x +
// Save the word count back out to a text file, causing evaluati
counts.saveAsTextFile(outputFile);
```

Example 2-11. Word count Scala application—don't worry about the details yet

```
// Create a Scala Spark Context.
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
// Load our input data.
val input =  sc.textFile(inputFile)
// Split it up into words.
val words = input.flatMap(line => line.split(" "))
// Transform into pairs and count.
val counts = words.map(word => (word, 1)).reduceByKey{case (x,
// Save the word count back out to a text file, causing evaluati
counts.saveAsTextFile(outputFile)
```

We can build these applications using very simple build files with both sbt (Example 2-12) and Maven (Example 2-13). We've marked the Spark Core dependency as `provided` so that, later on, when we use an assembly JAR we don't include the `spark-core` JAR, which is already on the classpath of the workers.

Example 2-12. sbt build file

```
name := "learning-spark-mini-example"

version := "0.0.1"

scalaVersion := "2.10.4"

// additional libraries
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.2.0" % "provided"
)
```

Example 2-13. Maven build file

```
<project>
  <groupId>com.oreilly.learningsparkexamples.mini</groupId>
  <artifactId>learning-spark-mini-example</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>example</name>
  <packaging>jar</packaging>
  <version>0.0.1</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.2.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <properties>
```

package our application into an assembly JAR. This is covered in more detail in Chapter 7.

Once we have our build defined, we can easily package and run our application using the `bin/spark-submit` script. The `spark-submit` script sets up a number of environment variables used by Spark. From the *mini-complete-example* directory we can build in both Scala (Example 2-14) and Java (Example 2-15).

Example 2-14. Scala build and run

```
sbt clean package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.scala.WordCount \
  ./target/...(as above) \
  ./README.md ./wordcounts
```

Example 2-15. Maven build and run

```
mvn clean && mvn compile && mvn package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.java.WordCount \
  ./target/learning-spark-mini-example-0.0.1.jar \
  ./README.md ./wordcounts
```

For even more detailed examples of linking applications to Spark, refer to the Quick Start Guide (http://spark.apache.org/docs/latest/quick-start.html) in the official Spark documentation. Chapter 7 covers packaging Spark applications in more detail.

Conclusion

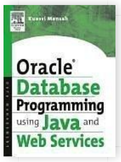
In this chapter, we have covered downloading Spark, running it locally on your laptop, and using it either interactively or from a standalone application. We gave a quick overview of the core concepts involved in programming with Spark: a driver program creates a SparkContext and RDDs, and then runs parallel operations on them. In the next chapter, we will dive more deeply into how RDDs operate.

PREV
1. Introduction to Data Analysis with Spark

5 MINS

R

BOOK SECTION



Web Services and SOA for DBA, Data Architects, and Others

from: Oracle Database Programming Using Java and Web Services by Kuassi Mensah

Released: July 2006

22 MINS

Oracle

BOOK SECTION

Monte Carlo II: Improving technique

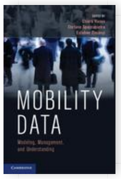
from: Computation and Modelling in Insurance and Finance by Erik Bølviken

Released: March 2014

66 MINS

Math & Science

BOOK SECTION



Trajectory Databases

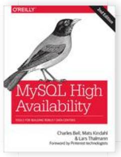
from: Mobility Data by Chiara Renso...

Released: August 2013

28 MINS

Linked Data

BOOK SECTION



Data Sharding

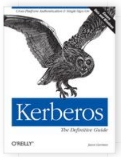
from: MySQL High Availability, 2nd Edition by Charles Bell...

Released: April 2014

71 MINS

MySQL

BOOK SECTION



Pieces of the Puzzle

from: Kerberos: The Definitive Guide by Jason Garman

Released: August 2003

7 MINS

Computer Networking