Community Experience Distilled

Fast Data Processing with Spark

High-speed distributed computing made easy with Spark

Holden Karau

[PACKT] open source* PUBLISHING
community experience distilled

Fast Data Processing with Spark

High-speed distributed computing made easy with Spark

Holden Karau

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Fast Data Processing with Spark

First published: October 2013

Production Reference: 1151013

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-706-8

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author
Holden Karau

Reviewers
Wayne Allan
Andrea Mostosi
Reynold Xin

Acquisition Editor
Kunal Parikh

Commissioning Editor
Shaon Basu

Technical Editors
Krutika Parab
Nadeem N. Bagban

Project Coordinator
Amey Sawant

Copy Editors
Brandt D'Mello
Kirti Pai
Lavina Pereira
Tanvi Gaitonde
Dipti Kapadia

Proofreader
Jonathan Todd

Indexer
Rekha Nair

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Holden Karau is a transgendered software developer from Canada currently living in San Francisco. Holden graduated from the University of Waterloo in 2009 with a Bachelors of Mathematics in Computer Science. She currently works as a Software Development Engineer at Google. She has worked at Foursquare, where she was introduced to Scala. She worked on search and classification problems at Amazon. Open Source development has been a passion of Holden's from a very young age, and a number of her projects have been covered on Slashdot. Outside of programming, she enjoys playing with fire, welding, and dancing. You can learn more at her website (`http://www.holdenkarau.com`), blog (`http://blog.holdenkarau.com`), and github (`https://github.com/holdenk`).

I'd like to thank everyone who helped review early versions of this book, especially Syed Albiz, Marc Burns, Peter J. J. MacDonald, Norbert Hu, and Noah Fiedel.

About the Reviewers

Andrea Mostosi is a passionate software developer. He started software development in 2003 at high school with a single-node LAMP stack and grew with it by adding more languages, components, and nodes. He graduated in Milan and worked on several web-related projects. He is currently working with data, trying to discover information hidden behind huge datasets.

I would like to thank my girlfriend, Khadija, who lovingly supports me in everything I do, and the people I collaborated with—for fun or for work—for everything they taught me. I'd also like to thank Packt Publishing and its staff for the opportunity to contribute to this book.

Reynold Xin is an Apache Spark committer and the lead developer for Shark and GraphX, two computation frameworks built on top of Spark. He is also a co-founder of Databricks which works on transforming large-scale data analysis through the Apache Spark platform. Before Databricks, he was pursuing a PhD in the UC Berkeley AMPLab, the birthplace of Spark.

Aside from engineering open source projects, he frequently speaks at Big Data academic and industrial conferences on topics related to databases, distributed systems, and data analytics. He also taught Palestinian and Israeli high-school students Android programming in his spare time.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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 `service@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.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

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

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface 1

Chapter 1: Installing Spark and Setting Up Your Cluster 5

- **Running Spark on a single machine** 7
- **Running Spark on EC2** 8
  - Running Spark on EC2 with the scripts 8
- **Deploying Spark on Elastic MapReduce** 13
- **Deploying Spark with Chef (opscode)** 14
- **Deploying Spark on Mesos** 15
- **Deploying Spark on YARN** 16
- **Deploying set of machines over SSH** 17
- **Links and references** 21
- **Summary** 22

Chapter 2: Using the Spark Shell 23

- **Loading a simple text file** 23
- **Using the Spark shell to run logistic regression** 25
- **Interactively loading data from S3** 27
- **Summary** 29

Chapter 3: Building and Running a Spark Application 31

- **Building your Spark project with sbt** 31
- **Building your Spark job with Maven** 35
- **Building your Spark job with something else** 37
- **Summary** 38

Chapter 4: Creating a SparkContext 39

- **Scala** 40
- **Java** 40
- **Shared Java and Scala APIs** 41
- **Python** 41

Links and references **42**
Summary **42**
Chapter 5: Loading and Saving Data in Spark **43**
RDDs **43**
Loading data into an RDD **44**
Saving your data **49**
Links and references **49**
Summary **50**
Chapter 6: Manipulating Your RDD **51**
Manipulating your RDD in Scala and Java **51**
Scala RDD functions 60
Functions for joining PairRDD functions 61
Other PairRDD functions 62
DoubleRDD functions 64
General RDD functions 64
Java RDD functions 66
Spark Java function classes 67
Common Java RDD functions 68
Methods for combining JavaPairRDD functions 69
JavaPairRDD functions 70
Manipulating your RDD in Python **71**
Standard RDD functions 73
PairRDD functions 75
Links and references **76**
Summary **76**
Chapter 7: Shark – Using Spark with Hive **77**
Why Hive/Shark? **77**
Installing Shark **78**
Running Shark **79**
Loading data **79**
Using Hive queries in a Spark program **80**
Links and references **83**
Summary **83**
Chapter 8: Testing **85**
Testing in Java and Scala **85**
Refactoring your code for testability 85
Testing interactions with SparkContext 88
Testing in Python **92**
Links and references **94**
Summary **94**

Chapter 9: Tips and Tricks **95**
Where to find logs? **95**
Concurrency limitations **95**
Memory usage and garbage collection **96**
Serialization **96**
IDE integration **97**
Using Spark with other languages **98**
A quick note on security **99**
Mailing lists **99**
Links and references **99**
Summary **100**
Index **101**

Preface

As programmers, we are frequently asked to solve problems or use data that is too much for a single machine to practically handle. Many frameworks exist to make writing web applications easier, but few exist to make writing distributed programs easier. The Spark project, which this book covers, makes it easy for you to write distributed applications in the language of your choice: Scala, Java, or Python.

What this book covers

Chapter 1, *Installing Spark and Setting Up Your Cluster*, covers how to install Spark on a variety of machines and set up a cluster—ranging from a local single-node deployment suitable for development work to a large cluster administered by a Chef to an EC2 cluster.

Chapter 2, *Using the Spark Shell*, gets you started running your first Spark jobs in an interactive mode. Spark shell is a useful debugging and rapid development tool and is especially handy when you are just getting started with Spark.

Chapter 3, *Building and Running a Spark Application*, covers how to build standalone jobs suitable for production use on a Spark cluster. While the Spark shell is a great tool for rapid prototyping, building standalone jobs is the way you will likely find most of your interaction with Spark to be.

Chapter 4, *Creating a SparkContext*, covers how to create a connection a Spark cluster. SparkContext is the entry point into the Spark cluster for your program.

Chapter 5, *Loading and Saving Your Data*, covers how to create and save RDDs (Resilient Distributed Datasets). Spark supports loading RDDs from any Hadoop data source.

Chapter 6, *Manipulating Your RDD*, covers how to do distributed work on your data with Spark. This chapter is the fun part.

Chapter 7, *Using Spark with Hive*, talks about how to set up Shark—a HiveQL-compatible system with Spark—and integrate Hive queries into your Spark jobs.

Chapter 8, *Testing*, looks at how to test your Spark jobs. Distributed tasks can be especially tricky to debug, which makes testing them all the more important.

Chapter 9, *Tips and Tricks*, looks at how to improve your Spark task.

What you need for this book

To get the most out of this book, you need some familiarity with Linux/Unix and knowledge of at least one of these programming languages: C++, Java, or Python. It helps if you have access to more than one machine or EC2 to get the most out of the distributed nature of Spark; however, it is certainly not required as Spark has an excellent standalone mode.

Who this book is for

This book is for any developer who wants to learn how to write effective distributed programs using the Spark project.

Conventions

In this book, you will find a number of styles of text 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: "The tarball file contains a bin directory that needs to be added to your path and `SCALA_HOME` should be set to the path where the tarball is extracted."

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

```
./run spark.examples.GroupByTest local[4]
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "by selecting **Key Pairs** under **Network & Security**".

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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

All of the example code from this book is hosted in three separate github repos:

- `https://github.com/holdenk/fastdataprocessingwithspark-sharkexamples`
- `https://github.com/holdenk/fastdataprocessingwithsparkexamples`
- `https://github.com/holdenk/chef-cookbook-spark`

Disclaimer

The opinions in this book are those of the author and not necessarily those any of my employers, past or present. The author has taken reasonable steps to ensure the example code is safe for use. You should verify the code yourself before using with important data. The author does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The author shall not be liable for any loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Installing Spark and Setting Up Your Cluster

This chapter will detail some common methods for setting up Spark. Spark on a single machine is excellent for testing, but you will also learn to use Spark's built-in deployment scripts to a dedicated cluster via SSH (Secure Shell). This chapter will also cover using Mesos, Yarn, Puppet, or Chef to deploy Spark. For cloud deployments of Spark, this chapter will look at EC2 (both traditional and EC2MR). Feel free to skip this chapter if you already have your local Spark instance installed and want to get straight to programming.

Regardless of how you are going to deploy Spark, you will want to get the latest version of Spark from `http://spark-project.org/download` (Version 0.7 as of this writing). For coders who live dangerously, try cloning the code directly from the repository `git://github.com/mesos/spark.git`. Both the source code and pre-built binaries are available. To interact with **Hadoop Distributed File System** (**HDFS**), you need to use a Spark version that is built against the same version of Hadoop as your cluster. For Version 0.7 of Spark, the pre-built package is built against Hadoop 1.0.4. If you are up for the challenge, it's recommended that you build against the source since it gives you the flexibility of choosing which HDFS version you want to support as well as apply patches. You will need the appropriate version of Scala installed and the matching JDK. For Version 0.7.1 of Spark, you require Scala 2.9.2 or a later 2.9 series release (2.9.3 works well). At the time of this writing, Ubuntu's LTS release (Precise) has Scala Version 2.9.1. Additionally, the current stable version has 2.9.2 and Fedora 18 has 2.9.2. Up-to-date package information can be found at `http://packages.ubuntu.com/search?keywords=scala`. The latest version of Scala is available from `http://scala-lang.org/download`. It is important to choose the version of Scala that matches the version requested by Spark, as Scala is a fast-evolving language.

The tarball file contains a bin directory that needs to be added to your path, and `SCALA_HOME` should be set to the path where the tarball file is extracted. Scala can be installed from source by running:

```
wget http://www.scala-lang.org/files/archive/scala-2.9.3.tgz && tar -xvf
scala-2.9.3.tgz && cd scala-2.9.3 && export PATH=`pwd`/bin:$PATH &&
export SCALA_HOME=`pwd`
```

You will probably want to add these to your `.bashrc` file or equivalent:

```
export PATH=`pwd`/bin:\$PATH
export SCALA_HOME=`pwd`
```

Spark is built with **sbt** (**simple build tool**, which is no longer very simple), and build times can be quite long when compiling Scala's source code. Don't worry if you don't have sbt installed; the build script will download the correct version for you.

On an admittedly under-powered core 2 laptop with an SSD, installing a fresh copy of Spark took about seven minutes. If you decide to build Version 0.7 from source, you would run:

```
wget http://www.spark-project.org/download-spark-0.7.0-sources-tgz &&
tar -xvf download-spark-0.7.0-sources-tgz && cd spark-0.7.0 && sbt/sbt
package
```

If you are going to use a version of HDFS that doesn't match the default version for your Spark instance, you will need to edit `project/SparkBuild.scala` and set `HADOOP_VERSION` to the corresponding version and recompile it with:

```
sbt/sbt clean compile
```

The sbt tool has made great progress with dependency resolution, but it's still strongly recommended for developers to do a clean build rather than an incremental build. This still doesn't get it quite right all the time.

Once you have started the build it's probably a good time for a break, such as getting a cup of coffee. If you find it stuck on a single line that says "Resolving [XYZ]...." for a long time (say five minutes), stop it and restart the `sbt/sbt` package.

If you can live with the restrictions (such as the fixed HDFS version), using the pre-built binary will get you up and running far quicker. To run the pre-built version, use the following command:

```
wget http://www.spark-project.org/download-spark-0.7.0-prebuilt-tgz &&
tar -xvf download-spark-0.7.0-prebuilt-tgz && cd spark-0.7.0
```

Spark has recently become a part of the Apache Incubator. As an application developer who uses Spark, the most visible changes will likely be the eventual renaming of the package to under the `org.apache` namespace.

Some of the useful links for references are as follows:

`http://spark-project.org/docs/latest`

`http://spark-project.org/download/`

`http://www.scala-lang.org`

Running Spark on a single machine

A single machine is the simplest use case for Spark. It is also a great way to sanity check your build. In the Spark directory, there is a shell script called run that can be used to launch a Spark job. Run takes the name of a Spark class and some arguments. There is a collection of sample Spark jobs in `./examples/src/main/scala/spark/examples/`.

All the sample programs take the parameter `master`, which can be the URL of a distributed cluster or `local[N]`, where `N` is the number of threads. To run `GroupByTest` locally with four threads, try the following command:

```
./run spark.examples.GroupByTest local[4]
```

If you get an error, as `SCALA_HOME` is not set, make sure your `SCALA_HOME` is set correctly. In bash, you can do this using the export `SCALA_HOME=[pathyouextractedscalato]`.

If you get the following error, it is likely you are using Scala 2.10, which is not supported by Spark 0.7:

```
[literal]"Exception in thread "main" java.lang.NoClassDefFoundError:
scala/reflect/ClassManifest"[/literal]
```

The Scala developers decided to rearrange some classes between 2.9 and 2.10 versions. You can either downgrade your version of Scala or see if the development build of Spark is ready to be built along with Scala 2.10.

Running Spark on EC2

There are many handy scripts to run Spark on EC2 in the `ec2` directory. These scripts can be used to run multiple Spark clusters, and even run on-the-spot instances. Spark can also be run on Elastic MapReduce (EMR). This is Amazon's solution for MapReduce cluster management, which gives you more flexibility around scaling instances.

Running Spark on EC2 with the scripts

To get started, you should make sure that you have EC2 enabled on your account by signing up for it at `https://portal.aws.amazon.com/gp/aws/manageYourAccount`. It is a good idea to generate a separate access key pair for your Spark cluster, which you can do at `https://portal.aws.amazon.com/gp/aws/securityCredentialsR`. You will also need to create an EC2 key pair, so that the Spark script can SSH to the launched machines; this can be done at `https://console.aws.amazon.com/ec2/home` by selecting **Key Pairs** under **Network & Security**. Remember that key pairs are created "per region", so you need to make sure you create your key pair in the same region as you intend to run your spark instances. Make sure to give it a name that you can remember (we will use `spark-keypair` in this chapter as its example key pair name) as you will need it for the scripts. You can also choose to upload your public SSH key instead of generating a new key. These are sensitive, so make sure that you keep them private. You also need to set your `AWS_ACCESS_KEY` and `AWS_SECRET_KEY` key pairs as environment variables for the Amazon EC2 scripts:

```
chmod 400 spark-keypair.pem
export AWS_ACCESS_KEY="..."
export AWS_SECRET_KEY="..."
```

You will find it useful to download the EC2 scripts provided by Amazon from `http://aws.amazon.com/developertools/Amazon-EC2/351`. Once you unzip the resulting ZIP file, you can add the `bin` folder to your `PATH` variable in a similar manner to what you did with the Spark `bin` folder:

```
wget http://s3.amazonaws.com/ec2-downloads/ec2-api-tools.zip
unzip ec2-api-tools.zip
cd ec2-api-tools-*
export EC2_HOME=`pwd`
export PATH=$PATH:`pwd`:/bin
```

The Spark EC2 script automatically creates a separate security group and firewall rules for the running Spark cluster. By default your Spark cluster will be universally accessible on port 8080, which is somewhat a poor form. Sadly, the `spark_ec2.py` script does not currently provide an easy way to restrict access to just your host. If you have a static IP address, I strongly recommend limiting the access in `spark_ec2.py`; simply replace all instances `0.0.0.0/0` with `[yourip]/32`. This will not affect intra-cluster communication, as all machines within a security group can talk to one another by default.

Next, try to launch a cluster on EC2:

```
./ec2/spark-ec2 -k spark-keypair -i pk-[....].pem -s 1 launch
myfirstcluster
```

If you get an error message such as "The requested Availability Zone is currently constrained and....", you can specify a different zone by passing in the `--zone` flag.

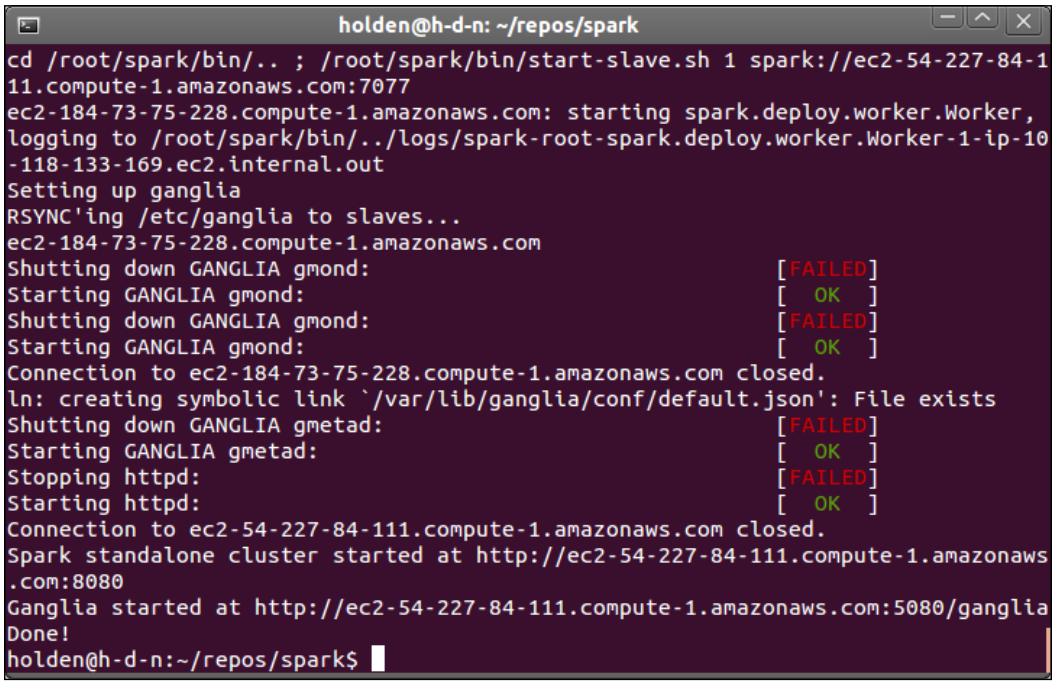
If you get an error about not being able to SSH to the master, make sure that only you have permission to read the private key, otherwise SSH will refuse to use it.

You may also encounter this error due to a race condition when the hosts report themselves as alive, but the Spark-`ec2` script cannot yet SSH to them. There is a fix for this issue pending in `https://github.com/mesos/spark/pull/555`. For now a temporary workaround, until the fix is available in the version of Spark you are using, is to simply let the cluster sleep an extra 120 seconds at the start of `setup_cluster`.

If you do get a transient error when launching a cluster, you can finish the launch process using the resume feature by running:

```
./ec2/spark-ec2 -i ~/spark-keypair.pem launch myfirstsparkcluster
--resume
```

If everything goes ok, you should see something like the following screenshot:



This will give you a bare-bones cluster with one master and one worker, with all the defaults on the default machine instance size. Next, verify that it has started up, and if your firewall rules were applied by going to the master on port 8080. You can see in the preceding screenshot that the name of the master is output at the end of the script.

Try running one of the example's jobs on your new cluster to make sure everything is ok:

```
sparkuser@h-d-n:~/repos/spark$ ssh -i ~/spark-keypair.pem root@ec2-107-22-48-231.compute-1.amazonaws.com
Last login: Sun Apr  7 03:00:20 2013 from 50-197-136-90-static.hfc.comcastbusiness.net
       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2012.03-release-notes/
There are 32 security update(s) out of 272 total update(s) available
```

```
Run "sudo yum update" to apply all updates.
Amazon Linux version 2013.03 is available.
[root@domU-12-31-39-16-B6-08 ~]# ls
ephemeral-hdfs  hive-0.9.0-bin  mesos  mesos-ec2  persistent-hdfs
scala-2.9.2  shark-0.2  spark  spark-ec2
[root@domU-12-31-39-16-B6-08 ~]# cd spark
[root@domU-12-31-39-16-B6-08 spark]# ./run spark.examples.GroupByTest
spark://`hostname`:7077
13/04/07 03:11:38 INFO slf4j.Slf4jEventHandler: Slf4jEventHandler started
13/04/07 03:11:39 INFO storage.BlockManagerMaster: Registered
BlockManagerMaster Actor
....
13/04/07 03:11:50 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 1.100294766 s
2000
```

Now that you've run a simple job on our EC2 cluster, it's time to configure your EC2 cluster for our Spark jobs. There are a number of options you can use to configure with the Spark-`ec2` script.

First, consider what instance types you may need. EC2 offers an ever-growing collection of instance types, and you can choose a different instance type for the master and the workers. The instance type has the most obvious impact on the performance of your spark cluster. If your work needs a lot of RAM, you should choose an instance with more RAM. You can specify the instance type with `--instance-type=(name of instance type)`. By default, the same instance type will be used for both the master and the workers. This can be wasteful if your computations are particularly intensive and the master isn't being heavily utilized. You can specify a different master instance type with `--master-instance-type=(name of instance)`.

EC2 also has GPU instance types that can be useful for workers, but would be completely wasted on the master. This text will cover working with Spark and GPUs later on; however, it is important to note that EC2 GPU performance may be lower than what you get while testing locally, due to the higher I/O overhead imposed by the hypervisor.

Downloading the example code

All of the example code from this book is hosted in three separate github repos:

- `https://github.com/holdenk/fastdataprocessingwithspark-sharkexamples`
- `https://github.com/holdenk/fastdataprocessingwithsparkexamples`
- `https://github.com/holdenk/chef-cookbook-spark`

Spark's EC2 scripts uses **AMI** (**Amazon Machine Images**) provided by the Spark team. These AMIs may not always be up-to-date with the latest version of Spark, and if you have custom patches (such as using a different version of HDFS) for Spark, they will not be included in the machine image. At present, the AMIs are also only available in the U.S. East region, so if you want to run it in a different region you will need to copy the AMIs or make your own AMIs in a different region.

To use Spark's EC2 scripts, you need to have an AMI available in your region. To copy the default Spark EC2 AMI to a new region, figure out what the latest Spark AMI is by looking at the `spark_ec2.py` script and seeing what URL the `LATEST_AMI_URL` points to and fetch it. For Spark 0.7, run the following command to get the latest AMI:

```
curl https://s3.amazonaws.com/mesos-images/ids/latest-spark-0.7
```

There is an ec2-copy-image script that you would hope provides the ability to copy the image, but sadly it doesn't work on images that you don't own. So you will need to launch an instance of the preceding AMI and snapshot it. You can describe the current image by running:

```
ec2-describe-images ami-a60193cf
```

This should show you that it is an **EBS**-based (**Elastic Block Store)** image, so you will need to follow EC2's instructions for creating EBS-based instances. Since you already have a script to launch the instance, you can just start an instance on an EC2 cluster and then snapshot it. You can find the instances you are running with:

```
ec2-describe-instances -H
```

You can copy the `i-[string]` instance name and save it for later use.

If you wanted to use a custom version of Spark or install any other tools or dependencies and have them available as part of our AMI, you should do that (or at least update the instance) before snapshotting.

```
ssh -i ~/spark-keypair.pem root@[hostname] "yum update"
```

Once you have your updates installed and any other customizations you want, you can go ahead and snapshot your instance with:

```
ec2-create-image -n "My customized Spark Instance" i-[instancename]
```

With the AMI name from the preceding code, you can launch your customized version of Spark by specifying the `[cmd]--ami[/cmd]` command-line argument. You can also copy this image to another region for use there:

```
ec2-copy-image -r [source-region] -s [ami] --region [target region]
```

This will give you a new AMI name, which you can use for launching your EC2 tasks. If you want to use a different AMI name, simply specify `--ami [aminame]`.

As of this writing, there was an issue with the default AMI and HDFS. You may need to update the version of Hadoop on the AMI, as it does not match the version that Spark was compiled for. You can refer to `https://spark-project.atlassian.net/browse/SPARK-683` for details.

Deploying Spark on Elastic MapReduce

In addition to Amazon's basic EC2 machine offering, Amazon offers a hosted MapReduce solution called Elastic MapReduce. Amazon provides a bootstrap script that simplifies the process of getting started using Spark on EMR. You can install the EMR tools from Amazon using the following command:

```
mkdir emr && cd emr && wget http://elasticmapreduce.s3.amazonaws.com/
elastic-mapreduce-ruby.zip && unzip *.zip
```

So that the EMR scripts can access your AWS account, you will want to create a `credentials.json` file:

```
{
    "access-id": "<Your AWS access id here>",
    "private-key": "<Your AWS secret access key here>",
    "key-pair": "<The name of your ec2 key-pair here>",
    "key-pair-file": "<path to the .pem file for your ec2 key pair
here>",
    "region": "<The region where you wish to launch your job flows
(e.g us-east-1)>"
}
```

Once you have the EMR tools installed, you can launch a Spark cluster by running:

```
elastic-mapreduce --create --alive --name "Spark/Shark Cluster" \
--bootstrap-action s3://elasticmapreduce/samples/spark/install-spark-
shark.sh \
--bootstrap-name "install Mesos/Spark/Shark" \
--ami-version 2.0  \
--instance-type m1.large --instance-count 2
```

This will give you a running EC2MR instance after about five to ten minutes. You can list the status of the cluster by running `elastic-mapreduce --list`. Once it outputs `j-[jobid]`, it is ready.

Some of the useful links that you can refer to are as follows:

- `http://aws.amazon.com/articles/4926593393724923`
- `http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-install.html`

Deploying Spark with Chef (opscode)

Chef is an open source automation platform that has become increasingly popular for deploying and managing both small and large clusters of machines. Chef can be used to control a traditional static fleet of machines, but can also be used with EC2 and other cloud providers. Chef uses cookbooks as the basic building blocks of configuration and can either be generic or site specific. If you have not used Chef before, a good tutorial for getting started with Chef can be found at `https://learnchef.opscode.com/`. You can use a generic Spark cookbook as the basis for setting up your cluster.

To get Spark working, you need to create a role for both the master and the workers, as well as configure the workers to connect to the master. Start by getting the cookbook from `https://github.com/holdenk/chef-cookbook-spark`. The bare minimum is setting the master hostname as master (so the worker nodes can connect) and the username so that Chef can install in the correct place. You will also need to either accept Sun's Java license or switch to an alternative JDK. Most of the settings that are available in `spark-env.sh` are also exposed through the cookbook's settings. You can see an explanation of the settings on configuring multiple hosts over SSH in the *Set of machines over SSH* section. The settings can be set per-role or you can modify the global defaults:

To create a role for the master with knife role, create `spark_master_role -e [editor]`. This will bring up a template role file that you can edit. For a simple master, set it to:

```
{
  "name": "spark_master_role",

  "description": "",
  "json_class": "Chef::Role",

  "default_attributes": {
    },
  "override_attributes": {
   "username":"spark",
   "group":"spark",
   "home":"/home/spark/sparkhome",
```

```
    "master_ip":"10.0.2.15",
  },
  "chef_type": "role",
  "run_list": [
    "recipe[spark::server]",
    "recipe[chef-client]",
  ],
  "env_run_lists": {
    },
}
```

Then create a role for the client in the same manner except instead of `spark::server`, use the `spark::client` recipe. Deploy the roles to the different hosts:

```
knife node run_list add master role[spark_master_role]
knife node run_list add worker role[spark_worker_role]
```

Then run `chef-client` on your nodes to update. Congrats, you now have a Spark cluster running!

Deploying Spark on Mesos

Mesos is a cluster management platform for running multiple distributed applications or frameworks on a cluster. Mesos can intelligently schedule and run Spark, Hadoop, and other frameworks concurrently on the same cluster. Spark can be run on Mesos either by scheduling individual jobs as separate Mesos tasks or running all of Spark as a single Mesos task. Mesos can quickly scale up to handle large clusters, beyond the size of which you would want to manage, with plain old SSH scripts. It was originally created at UC Berkley as a research project; it is currently undergoing Apache incubation and is actively used by Twitter.

To get started with Mesos, you can download the latest version from `http://mesos.apache.org/downloads/` and unpack the ZIP files. Mesos has a number of different configuration scripts you can use; for an Ubuntu installation use `configure.ubuntu-lucid-64`, and for other cases the Mesos `README` file will point you at which configuration file to use. In addition to the requirements of Spark, you will need to ensure that you have the Python C header files installed (`python-dev` on Debian systems) or pass `--disable-python` to the configured script. Since Mesos needs to be installed on all the machines, you may find it easier to configure Mesos to install somewhere other than the root, most easily alongside your Spark installation as follows:

```
./configure --prefix=/home/sparkuser/mesos && make && make check && make
install
```

Much like with the configuration of Spark in standalone mode with Mesos, you need to make sure the different Mesos nodes can find one another. Start with adding `mesossprefix/var/mesos/deploy/masters` to the hostname of the master, and then adding each worker hostname to `mesossprefix/var/mesos/deploy/slaves`. Then you will want to point the workers at the master (and possibly set some other values) in `mesossprefix/var/mesos/conf/mesos.conf`.

Once you have Mesos built, it's time to configure Spark to work with Mesos. This is as simple as copying the `conf/spark-env.sh.template` to `conf/spark-env.sh`, and updating `MESOS_NATIVE_LIBRARY` to point to the path where Mesos is installed. You can find more information about the different settings in `spark-env.sh` in the table shown in the next section.

You will need to install both Mesos on Spark on all the machines in your cluster. Once both Mesos and Spark are configured, you can copy the build to all the machines using `pscp` as shown in the following command:

```
pscp -v -r -h  -l sparkuser ./mesos /home/sparkuser/mesos
```

You can then start your Mesos clusters by using `mesosprefix/sbin/mesos-start-cluster.sh`, and schedule your Spark on Mesos by using `mesos://[host]:5050` as the master.

Deploying Spark on YARN

YARN is Apache Hadoop's NextGen MapReduce. The Spark project provides an easy way to schedule jobs on YARN once you have a Spark assembly built. It is important that the Spark job you create uses a standalone master URL. The example Spark applications all read the master URL from the command-line arguments, so specify `--args` standalone.

To run the same example as in the SSH section, do the following:

```
sbt/sbt assembly #Build the assembly

SPARK_JAR=./core/target/spark-core-assembly-0.7.0.jar ./run spark.deploy.
yarn.Client --jar examples/target/scala-2.9.2/spark-examples_2.9.2-
0.7.0.jar --class spark.examples.GroupByTest --args standalone --num-
workers 2 --worker-memory 1g --worker-cores 1
```

Deploying set of machines over SSH

If you have a set of machines without any existing cluster management software, you can deploy Spark over SSH with some handy scripts. This method is known as "standalone mode" in the Spark documentation. An individual master and worker can be started by `./run spark.deploy.master.Master` and `./run spark.deploy.worker.Worker spark://MASTERIP:PORT` respectively. The default port for the master is 8080. It's likely that you don't want to go to each of your machines and run these commands by hand; there are a number of helper scripts in `bin/` to help you run your servers.

A prerequisite for using any of the scripts is having a password-less SSH access setup from the master to all the worker machines. You probably want to create a new user for running Spark on the machines and lock it down. This book uses the username `sparkuser`. On your master machine, you can run `ssh-keygen` to generate the SSH key and make sure that you do not set a password. Once you have generated the key, add the public one (if you generated an RSA key it would be stored in `~/.ssh/id_rsa.pub` by default) to `~/.ssh/authorized_keys2` on each of the hosts.

The Spark administration scripts require that your username matches. If this isn't the case, you can configure an alternative username in your `~/.ssh/config`.

Now that you have SSH access to the machines set up, it is time to configure Spark. There is a simple template in `[filepath]conf/spark-env.sh.template[/filepath]` that you should copy to `[filepath]conf/spark-env.sh[/filepath]`. You will need to set the `SCALA_HOME` variable to the path where you extracted Scala to. You may also find it useful to set some (or all) of the following environment variables:

Name	Purpose	Default
`MESOS_NATIVE_LIBRARY`	Point to match where Mesos is located	`None`
`SCALA_HOME`	Point to where you extracted Scala	`None`, must be set
`SPARK_MASTER_IP`	The IP address for the master to listen on and the IP address for the workers to connect to port #	The result of running hostname

Name	Purpose	Default
`SPARK_MASTER_PORT`	The port # for the Spark master to listen on	`7077`
`SPARK_MASTER_WEBUI_PORT`	The port # of the web UI on the master	`8080`
`SPARK_WORKER_CORES`	The number of cores to use	All of them
`SPARK_WORKER_MEMORY`	The amount of memory to use	Max of system memory - (minus) 1 GB (or 512 MB)
`SPARK_WORKER_PORT`	The port # on which the worker runs on	`random`
`SPARK_WEBUI_PORT`	The port # on which the worker web UI runs on	`8081`
`SPARK_WORKER_DIR`	The location where to store files from the worker	`SPARK_HOME/work_dir`

Once you have your configuration all done, it's time to get your cluster up and running. You will want to copy the version of Spark and the configurations you have built to all of your machines. You may find it useful to install PSSH, a set of parallel SSH tools including PCSP. The PSCP application makes it easy to SCP (securely copy files) to a number of target hosts, although it will take a while, such as:

```
pscp -v -r -h conf/slaves -l sparkuser ../spark-0.7.0 ~/
```

If you end up changing the configuration, you need to distribute the configuration to all the workers, such as:

```
pscp -v -r -h conf/slaves -l sparkuser conf/spark-env.sh ~/
spark-0.7.0/conf/spark-env.sh
```

If you use a shared NFS on your cluster—although by default Spark names logfiles and similar with the shared names—you should configure a separate worker directory otherwise they will be configured to write to the same place. If you want to have your worker directories on the shared NFS, consider adding `` `hostname` ``, for example, `SPARK_WORKER_DIR=~/work-`hostname``.

You should also consider having your logfiles go to a scratch directory for better performance.

If you don't have Scala installed on the remote machines yet, you can also use `pssh` to set it up:

```
pssh -P -i -h conf/slaves -l sparkuser "wget http://www.scala-lang.org/
downloads/distrib/files/scala-2.9.3.tgz && tar -xvf scala-2.9.3.tgz && cd
scala-2.9.3 && export PATH=$PATH:`pwd`/bin && export SCALA_HOME=`pwd` &&
echo \"export PATH=`pwd`/bin:\\\\$PATH && export SCALA_HOME=`pwd`\" >>
~/.bashrc"
```

Now you are ready to start the cluster. It is important to note that `start-all` and `start-master` both assume they are being run on the node, which is the master for the cluster. The start scripts all daemonize, so you don't have to worry about running them in a screen.

```
ssh master bin/start-all.sh
```

If you get a class not found error, such as `java.lang.NoClassDefFoundError: scala/ScalaObject`, check to make sure that you have Scala installed on that worker host and that the `SCALA_HOME` is set correctly.

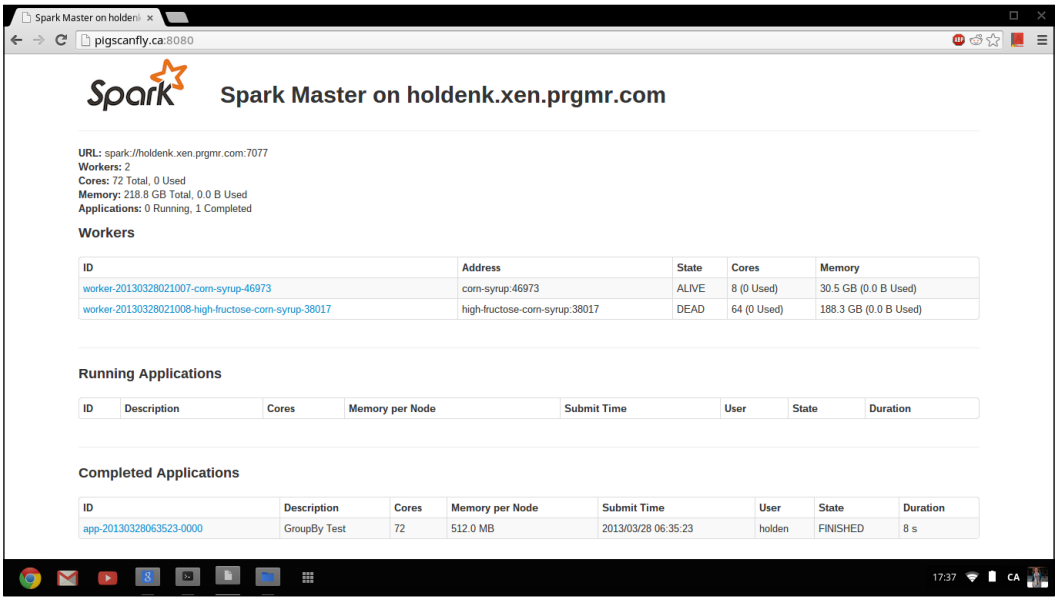
The Spark scripts assume that your master has Spark installed as the same directory as your workers. If this is not the case, you should edit `bin/spark-config.sh` and set it to the appropriate directories.

The commands provided by Spark to help you administer your cluster are in the following table:

Command	Use
`bin/slaves.sh <command>`	Runs the provided command on all the worker hosts. For example, `bin/slave.sh` uptime will show how long each of the worker hosts have been up.
`bin/start-all.sh`	Starts the master and all the worker hosts. It must be run on the master.
`bin/start-master.sh`	Starts the master host. Must be run on the master.
`bin/start-slaves.sh`	Starts the worker hosts.
`bin/start-slave.sh`	Start a specific worker.
`bin/stop-all.sh`	Stops master and workers.

Command	Use
`bin/stop-master.sh`	Stops the master.
`bin/stop-slaves.sh`	Stops all the workers.

You now have a running Spark cluster, as shown in the following screenshot. There is a handy web UI on the master running on port 8080; you should visit and switch on all the workers on port 8081. The web UI contains such helpful information as the current workers, and current and past jobs.



Now that you have a cluster up and running let's actually do something with it. As with the single host example, you can use the provided run script to run Spark commands. All the examples listed in `examples/src/main/scala/spark/examples/` take a parameter, `master`, which points them to the master machine. Assuming you are on the master host you could run them like this:

```
./run spark.examples.GroupByTest spark://`hostname`:7077
```

If you run into an issue with `java.lang.UnsupportedClassVersionError`, you may need to update your JDK or recompile Spark if you grabbed the binary version. Version 0.7 was compiled with JDK 1.7 as the target. You can check the version of the JRE targeted by Spark with:

```
java -verbose -classpath ./core/target/scala-2.9.2/
classes/
spark.SparkFiles | head -n 20
```

Version 49 is JDK1.5, Version 50 is JDK1.6, and Version 60 is JDK1.7.

If you can't connect to the localhost, make sure that you've configured your master to listen to all the IP addresses (or if you don't want to replace the localhost with the IP address configured to listen too).

If everything has worked correctly, you will see a lot of log messages output to `stdout` something along the lines of:

```
13/03/28 06:35:31 INFO spark.SparkContext: Job finished: count at
GroupByTest.scala:35, took 2.482816756 s
2000
```

Links and references

Some of the useful links are as follows:

- `http://archive09.linux.com/feature/151340`
- `http://spark-project.org/docs/latest/spark-standalone.html`
- `https://github.com/mesos/spark/blob/master/core/src/main/scala/spark/deploy/worker/WorkerArguments.scala`
- `http://bickson.blogspot.com/2012/10/deploying-graphlabsparkmesos-cluster-on.html`
- `http://www.ibm.com/developerworks/library/os-spark/`
- `http://mesos.apache.org/`
- `http://aws.amazon.com/articles/Elastic-MapReduce/4926593393724923`
- `http://spark-project.org/docs/latest/ec2-scripts.html`

Summary

In this chapter, we have installed Spark on our machine for local development and also set up on our cluster, so we are ready to run the applications that we write. In the next chapter, we will learn to use the Spark shell.

2
Using the Spark Shell

The Spark shell is a wonderful tool for rapid prototyping with Spark. It helps to be familiar with Scala, but it isn't necessary when using this tool. The Spark shell allows you to query and interact with the Spark cluster. This can be great for debugging or for just trying things out. The previous chapter should have gotten you to the point of having a Spark instance running, so now all you need to do is start your Spark shell, and point it at your running index with the following command:

```
MASTER=spark://`hostname`:7077 ./spark-shell
```

If you are running Spark in local mode and don't have a Spark instance already running, you can just run the preceding command without the `MASTER=` part. This will run with only one thread, hence to run multiple threads you can specify `local[n]`.

Loading a simple text file

When running a Spark shell and connecting to an existing cluster, you should see something specifying the `app ID` like `Connected to Spark cluster with app ID app-20130330015119-0001`. The `app ID` will match the application entry as shown in the web UI under running applications (by default, it would be viewable on port 8080). You can start by downloading a dataset to use for some experimentation. There are a number of datasets that are put together for *The Elements of Statistical Learning*, which are in a very convenient form for use. Grab the spam dataset using the following command:

```
wget http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/spam.data
```

Now load it as a text file into Spark with the following command inside your Spark shell:

```
scala> val inFile = sc.textFile("./spam.data")
```

This loads the `spam.data` file into Spark with each line being a separate entry in the **RDD** (**Resilient Distributed Datasets**).

Note that if you've connected to a Spark master, it's possible that it will attempt to load the file on one of the different machines in the cluster, so make sure it's available on all the cluster machines. In general, in future you will want to put your data in HDFS, S3, or similar file systems to avoid this problem. In a local mode, you can just load the file directly, for example, `sc.textFile([filepah])`. To make a file available across all the machines, you can also use the `addFile` function on the SparkContext by writing the following code:

```
scala> import spark.SparkFiles;
scala> val file = sc.addFile("spam.data")
scala> val inFile = sc.textFile(SparkFiles.get("spam.data"))
```

Just like most shells, the Spark shell has a command history. You can press the up arrow key to get to the previous commands. Getting tired of typing or not sure what method you want to call on an object? Press *Tab*, and the Spark shell will autocomplete the line of code as best as it can.

For this example, the RDD with each line as an individual string isn't very useful, as our data input is actually represented as space-separated numerical information. Map over the RDD, and quickly convert it to a usable format (note that `_.toDouble` is the same as `x => x.toDouble`):

```
scala> val nums = inFile.map(x => x.split(' ').map(_.toDouble))
```

Verify that this is what we want by inspecting some elements in the `nums` RDD and comparing them against the original string RDD. Take a look at the first element of each RDD by calling `.first()` on the RDDs:

```
scala> inFile.first()
[...]
res2: String = 0 0.64 0.64 0 0.32 0 0 0 0 0 0 0.64 0 0 0 0.32 0 1.29
1.93 0 0.96 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0.778 0 0 3.756 61 278 1

scala> nums.first()
[...]
res3: Array[Double] = Array(0.0, 0.64, 0.64, 0.0, 0.32, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.64, 0.0, 0.0, 0.0, 0.32, 0.0, 1.29, 1.93, 0.0, 0.96,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.778, 0.0, 0.0, 3.756, 61.0, 278.0, 1.0)
```

Using the Spark shell to run logistic regression

When you run a command and have not specified a left-hand side (that is, leaving out the `val x` of `val x = y`), the Spark shell will print the value along with `res[number]`. The `res[number]` function can be used as if we had written `val res[number] = y`. Now that you have the data in a more usable format, try to do something cool with it! Use Spark to run logistic regression over the dataset as follows:

```
scala> import spark.util.Vector
import spark.util.Vector

scala> case class DataPoint(x: Vector, y: Double)
defined class DataPoint

scala> def parsePoint(x: Array[Double]): DataPoint = {
      DataPoint(new Vector(x.slice(0,x.size-2)) , x(x.size-1))
      }
parsePoint: (x: Array[Double])this.DataPoint

scala> val points = nums.map(parsePoint(_))
points: spark.RDD[this.DataPoint] = MappedRDD[3] at map at
<console>:24

scala> import java.util.Random
import java.util.Random

scala> val rand = new Random(53)
rand: java.util.Random = java.util.Random@3f4c24
scala> var w = Vector(nums.first.size-2, _ => rand.nextDouble)
13/03/31 00:57:30 INFO spark.SparkContext: Starting job: first at
<console>:20
...
13/03/31 00:57:30 INFO spark.SparkContext: Job finished: first at
<console>:20, took 0.01272858 s
w: spark.util.Vector = (0.7290865701603526, 0.8009687428076777,
0.6136632797111822, 0.9783178194773176, 0.3719683631485643,
0.46409291255379836, 0.5340172959927323, 0.04034252433669905,
0.3074428389716637, 0.8537414030626244, 0.8415816118493813,
0.719935849109521, 0.2431646830671812, 0.17139348575456848,
0.5005137792223062, 0.8915164469396641, 0.7679331873447098,
0.7887571495335223, 0.7263187438977023, 0.40877063468941244,
0.7794519914671199, 0.1651264689613885, 0.1807006937030201,
```

```
0.3227972103818231, 0.2777324549716147, 0.20466985600105037,
0.5823059390134582, 0.4489508737465665, 0.44030858771499415,
0.6419366305419459, 0.5191533842209496, 0.43170678028084863,
0.9237523536173182, 0.5175019655845213, 0.47999523211827544,
0.25862648071479444, 0.020548000101787922, 0.18555332739714137, 0....

scala> val iterations = 100
iterations: Int = 100

scala> import scala.math._

scala> for (i <- 1 to iterations) {
        val gradient = points.map(p =>
          (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
        ).reduce(_ + _)
        w -= gradient
      }
[....]

scala> w
res27: spark.util.Vector = (0.2912515190246098, 1.05257972144256,
1.1620192443948825, 0.764385365541841, 1.3340446477767611,
0.6142105091995632, 0.8561985593740342, 0.7221556020229336,
0.40692442223198366, 0.8025693176035453, 0.7013618380649754,
0.943828424041885, 0.4009868306348856, 0.6287356973527756,
0.3675755379524898, 1.2488466496117185, 0.8557220216380228,
0.7633511642942988, 6.389181646047163, 1.43344096405385,
1.729216408954399, 0.4079709812689015, 0.3706358251228279,
0.8683036382227542, 0.36992902312625897, 0.3918455398419239,
0.2840295056632881, 0.7757126171768894, 0.4564171647415838,
0.6960856181900357, 0.6556402580635656, 0.060307680034745986,
0.31278587054264356, 0.9273189009376189, 0.0538302050535121,
0.545536066902774, 0.9298009485403773, 0.922750704590723,
0.072339496591
```

If things went well, you just used Spark to run logistic regression. Awsome! We have just done a number of things: we have defined a class, we have created an RDD, and we have also created a function. As you can see the Spark shell is quite powerful. Much of the power comes from it being based on the Scala REPL (the Scala interactive shell), so it inherits all the power of the Scala REPL (Read-Evaluate-Print Loop). That being said, most of the time you will probably want to work with a more traditionally compiled code rather than working in the REPL environment.

Interactively loading data from S3

Now, let's try a second exercise with the Spark shell. As part of Amazon's EMR Spark support, it has provided some handy sample data of Wikipedia's traffic statistics in S3 in the format that Spark can use. To access the data, you first need to set your AWS access credentials as shell's parameters. For instructions on signing up for EC2 and setting up the shell parameters, see the *Running Spark on EC2* with the scripts section in *Chapter 1*, *Installing Spark and Setting Up Your Cluster* (S3 access requires additional keys `fs.s3n.awsAccessKeyId/awsSecretAccessKey` or using the `s3n://user:pw@` syntax). Once that's done, load the S3 data and take a look at the first line:

```
scala> val file = sc.textFile("s3n://bigdatademo/sample/wiki/")
13/04/21 21:26:14 INFO storage.MemoryStore: ensureFreeSpace(37539)
called with curMem=37531, maxMem=339585269
13/04/21 21:26:14 INFO storage.MemoryStore: Block broadcast_1 stored
as values to memory (estimated size 36.7 KB, free 323.8 MB)
file: spark.RDD[String] = MappedRDD[3] at textFile at <console>:12

scala> file.take(1)
13/04/21 21:26:17 INFO mapred.FileInputFormat: Total input paths to
process : 1
...
13/04/21 21:26:17 INFO spark.SparkContext: Job finished: take at
<console>:15, took 0.533611079 s
res1: Array[String] = Array(aa.b Pecial:Listusers/sysop 1 4695)
```

You don't need to set your AWS credentials as shell's parameters; the general form of the S3 path is `s3n://<AWS ACCESS ID>:<AWS SECRET>@bucket/path`. It's important to take a look at the first line of data because unless we force Spark to materialize something with the data, it won't actually bother to load it. It is useful to note that Amazon provided a small sample dataset to get started with. The data is pulled from a much larger set at `http://aws.amazon.com/datasets/4182`. This practice can be quite useful, when developing in interactive mode, since you want the fast feedback of your jobs completing quickly. If your sample data was too big and your executions were taking too long, you could quickly slim down the RDD by using the `sample` functionality built into the Spark shell:

```
scala> val seed  = (100*math.random).toInt
seed: Int = 8
scala> file.sample(false,1/10.,seed)
res10: spark.RDD[String] = SampledRDD[4] at sample at <console>:17

//If you wanted to rerun on the sampled data later, you could write it
back to S3
scala> res10.saveAsTextFile("s3n://mysparkbucket/test")
```

```
13/04/21 22:46:18 INFO spark.PairRDDFunctions: Saving as hadoop file
of type (NullWritable, Text)
....
13/04/21 22:47:46 INFO spark.SparkContext: Job finished:
saveAsTextFile at <console>:19, took 87.462236222 s
```

Now that you have the data loaded, find the most popular articles in a sample. First, parse the data separating it into name and count. Second, as there can be multiple entries with the same name, reduce the data by the key summing the counts. Finally, we swap the key/value so that when we sort by key, we get back the highest count item as follows:

```
scala> val parsed = file.sample(false,1/10.,seed).map(x => x.split("
")).map(x => (x(1), x(2).toInt))
parsed: spark.RDD[(java.lang.String, Int)] = MappedRDD[5] at map at
<console>:16

scala> val reduced = parsed.reduceByKey(_+_)
13/04/21 23:21:49 WARN util.NativeCodeLoader: Unable to load native-
hadoop library for your platform... using builtin-java classes where
applicable
13/04/21 23:21:49 WARN snappy.LoadSnappy: Snappy native library not
loaded
13/04/21 23:21:50 INFO mapred.FileInputFormat: Total input paths to
process : 1
reduced: spark.RDD[(java.lang.String, Int)] = MapPartitionsRDD[8] at
reduceByKey at <console>:18

scala> val countThenTitle = reduced.map(x => (x._2, x._1))
countThenTitle: spark.RDD[(Int, java.lang.String)] = MappedRDD[9] at
map at <console>:20

scala> countThenTitle.sortByKey(false).take(10)
13/04/21 23:22:08 INFO spark.SparkContext: Starting job: take at
<console>:23
....
13/04/21 23:23:15 INFO spark.SparkContext: Job finished: take at
<console>:23, took 66.815676564 s
res1: Array[(Int, java.lang.String)] = Array((213652,Main_Page),
(14851,Special:Search), (9528,Special:Export/Can_You_Hear_Me),
(6454,Wikipedia:Hauptseite), (4189,Special:Watchlist), (3520,%E7
%89%B9%E5%88%A5:%E3%81%8A%E3%81%BE%E3%81%8B%E3%81%9B%E8%A1%A8%E7
%A4%BA), (2857,Special:AutoLogin), (2416,P%C3%A1gina_principal),
(1990,Survivor_(TV_series)), (1953,Asperger_syndrome))
```

You can also work with Spark interactively in Python by running `./pyspark`.

Summary

In this chapter, you have learned how to start the Spark shell, load our data, and we did a few simple things through a hands-on machine-learning approach. Now that you've seen how Spark's interactive console works, it's time to see how to build Spark jobs in a more traditional and persistent environment in the subsequent chapter.

3
Building and Running a Spark Application

Using Spark in an interactive mode with the Spark shell has limited permanence and does not work in Java. Building Spark jobs is a bit trickier than building a normal application as all the dependencies have to be available on all the machines that are in your cluster. This chapter will cover building a Java and Scala Spark job with Maven or sbt and Spark jobs with a non-maven-aware build system.

Building your Spark project with sbt

The sbt tool is a popular build tool for Scala that supports building both Scala and Java code. Building Spark projects with sbt is one of the easiest options because Spark itself is built with sbt. It makes it easy to bring in dependencies (which is especially useful for Spark) as well as package everything into a single deployable/JAR file. The current normal method of building packages that use sbt is to use a shell script that bootstraps the specific version of sbt that your project uses, making installation simpler.

As a first step, take a Spark job that already works and go through the process of creating a build file for it. In the `spark` directory, begin by copying the `GroupByTest` example into a new directory as follows:

```
mkdir -p example-scala-build/src/main/scala/spark/examples/

cp -af sbt example-scala-build/

cp examples/src/main/scala/spark/examples/GroupByTest.scala example-scala-build/src/main/scala/spark/examples/
```

Since you are going to ship your JAR file to the other machines, you will want to ensure that all the dependencies are included in them. You can either add a bunch of JAR files or use a handy sbt plugin called `sbt-assembly` to group everything into a single JAR file. If you don't have a bunch of transitive dependencies, you may decide that using the assembly extension isn't useful for your project. Instead of using `sbt-assembly`, you probably want to run `sbt/sbt assembly` in the Spark project and add the resulting JAR file `core/target/spark-core-assembly-0.7.0.jar` to your classpath. The `sbt-assembly` package is a great tool to avoid having to manually manage a large number of JAR files. To add the assembly extension to your build, add the following code to `project/plugins.sbt`:

```
resolvers += Resolver.url("artifactory",
url("http://scalasbt.artifactoryonline.com/scalasbt/
sbt-plugin-releases"))(Resolver.ivyStylePatterns)

resolvers += "Typesafe Repository" at
"http://repo.typesafe.com/typesafe/releases/"

resolvers += "Spray Repository" at "http://repo.spray.cc/"
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.8.7")
```

Resolvers are used by sbt so that it can find where a package is; you can think of this as similar to specifying an additional APT **PPA** (**Personal Package Archive**) source, except it only applies to the one package that you are trying to build. If you load up the resolver URLs in your browser, most of them have directory listing turned on, so you can see what packages are provided by the resolver. These resolvers point to web URLs, but there are also resolvers for local paths, which can be useful during development. The `addSbtPlugin` directive is deceptively simple; it says to include the `sbt-assembly` package from `com.eed3si9n` at Version `0.8.7` and implicitly adds the Scala version and sbt version. Make sure to run `sbt reload clean update` to install new plugins.

The following is the build file for one of the `GroupByTest.scala` examples as if it were being built on its own; insert the following code in `./build.sbt`:

```
//Next two lines only needed if you decide to use the assembly plugin
import AssemblyKeys._
assemblySettings

scalaVersion := "2.9.2"

name := "groupbytest"

libraryDependencies ++= Seq(
```

```
  "org.spark-project" % "spark-core_2.9.2" % "0.7.0"
)

resolvers ++= Seq(
  "JBoss Repository" at
  "http://repository.jboss.org/nexus/content/
   repositories/releases/",
  "Spray Repository" at "http://repo.spray.cc/",
  "Cloudera Repository" at
  "https://repository.cloudera.com/artifactory/cloudera-repos/",
  "Akka Repository" at "http://repo.akka.io/releases/",
  "Twitter4J Repository" at "http://twitter4j.org/maven2/"
)
//Only include if using assembly
mergeStrategy in assembly <<= (mergeStrategy in assembly) {
  (old) =>
  {
    case PathList("javax", "servlet", xs @ _*) => MergeStrategy.first
    case PathList("org", "apache", xs @ _*) => MergeStrategy.first
    case "about.html"  => MergeStrategy.rename
    case x => old(x)
  }
}
```

As you can see, the build file is similar in format to `plugins.sbt`. There are a few unique things about this build file that are worth mentioning. Just as we did with the plugin file, you also need to add a number of resolvers so that sbt can find all the dependencies. Note that we are including it as `"org.spark-project" % "spark-core_2.9.2" % "0.7.0"` rather than using `"org.spark-project" %% "spark-core" % "0.7.0"`. If possible, you should try to use the `%%` format, which automatically adds the Scala version. Another unique part of this build file is the use of `mergeStrategy`. Since multiple dependencies can define the same files, when you merge everything into a single JAR file, you need to tell the plugin how to handle it. It is a fairly simple build file other than the merge strategy and manually specifying the Scala version of Spark that you are using.

If you have a different JDK on the master than JRE on the workers, you may want to switch the target JDK by adding the following code to your build file:

```
javacOptions ++= Seq("-target", "1.6")
```

Now that your build file is defined, build your `GroupByTest` Spark job:

```
sbt/sbt clean compile package
```

This will produce `target/scala-2.9.2/groupbytest_2.9.2-0.1-SNAPSHOT.jar`.

Run `sbt/sbt assembly` in the `spark` directory to make sure you have the Spark assembly available to your classpaths. The example requires a pointer to where Spark is using `SPARK_HOME` and where the `jar` example is using `SPARK_EXAMPLES_JAR`. We also need to specify the classpath that we built to Scala locally with `-cp`. We can then run the following example:

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/scala-2.9.2/
groupbytest-assembly-0.1-SNAPSHOT.jar"  scala -cp/users/
sparkuser/spark-0.7.0/example-scala-build/target/scala-2.9.2/
groupbytest_2.9.2-0.1-SNAPSHOT.jar:/users/sparkuser/spark-0.7.0/
core/target/
spark-core-assembly-0.7.0.jar spark.examples.GroupByTest local[1]
```

If you have decided to build all of your dependencies into a single JAR file with the assembly plugin, we need to call it like this:

```
sbt/sbt assembly
```

This will produce an assembly snapshot at `target/scala-2.9.2/groupbytest-assembly-0.1-SNAPSHOT.jar`, which you can then run in a very similar manner, simply without `spark-core-assembly`:

```
SPARK_HOME="../" \ SPARK_EXAMPLES_JAR="./target/scala-2.9.2/groupbytest-
assembly-0.1-SNAPSHOT.jar" \
 scala -cp /users/sparkuser/spark-0.7.0/example-scala-build/target/
scala-2.9.2/groupbytest-assembly-0.1-SNAPSHOT.jar spark.examples.
GroupByTest local[1]
```

You may run into merge issues with sbt assembly if things have changed; a quick search over the web will probably provide better current guidance than anything that could be written taking guesses about future merge problems. In general, `MergeStategy.first` should work.

Your success for the preceding code may have given you a false sense of security. Since `sbt` will resolve security from the local cache, the `deps` package that was brought in by another project could mean that the code builds on one machine and not others. Delete your local ivy cache and run `sbt clean` to make sure. If some files fail to download, try looking at Spark's list of resolvers and adding any missing ones to your `build.sbt`.

Some of the following links useful for referencing are as follows:

- `http://www.scala-sbt.org/`
- `https://github.com/sbt/sbt-assembly`
- `http://spark-project.org/docs/latest/scala-programming-guide.html`

Building your Spark job with Maven

Maven is an open source Apache project that builds Spark jobs in Java or Scala. As with sbt, you can include the Spark dependency through Maven, simplifying our build process. As with sbt, Maven has the ability to bundle Spark and all of our dependencies, in a single JAR file using a plugin or build Spark as a monolithic JAR using `sbt/sbt assembly` for inclusion.

To illustrate the build process for Spark jobs with Maven, this section will use Java as an example since Maven is more commonly used to build Java tasks. As a first step, let's take a Spark job that already works and go through the process of creating a build file for it. We can start by copying the `GroupByTest` example into a new directory and generating the Maven template as follows:

```
mkdir example-java-build/; cd example-java-build
mvn archetype:generate \
   -DarchetypeGroupId=org.apache.maven.archetypes \
   -DgroupId=spark.examples \
   -DartifactId=JavaWordCount \
   -Dfilter=org.apache.maven.archetypes:maven-archetype-quickstart
cp ../examples/src/main/java/spark/examples/JavaWordCount.java
JavaWordCount/src/main/java/spark/examples/JavaWordCount.java
```

Next, update your Maven `pom.xml` to include information about the version of Spark we are using. Also, since the example file we are working with requires JDK 1.5, we will need to update the Java version that Maven is configured to use; at the time of writing, it defaults to 1.3. In between the `<project>` tags, we will need to add the following code:

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
```

```
        <scope>test</scope>
      </dependency>
      <dependency>
        <groupId>org.spark-project</groupId>
        <artifactId>spark-core_2.9.2</artifactId>
        <version>0.7.0</version>
      </dependency>
    </dependencies>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.5</source>
            <target>1.5</target>
          </configuration>
        </plugin>
      </plugins>
    </build>
```

We can now build our `jar` with the `maven` package, which can be run using the following commands:

```
SPARK_HOME="../"  SPARK_EXAMPLES_JAR="./target/JavaWordCount-1.0-
SNAPSHOT.jar"  java -cp ./target/JavaWordCount-1.0-SNAPSHOT.jar:../../
core/target/spark-core-assembly-0.7.0.jar spark.examples.JavaWordCount
local[1] ../../README
```

As with sbt, we can use a plugin to include all the dependencies in our JAR file. In between the `<plugins>` tags, add the following code:

```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>1.7</version>
    <configuration>
      <!-- This transform is used so that merging of akka configuration
  files works -->
      <transformers>
        <transformer implementation=
    "org.apache.maven.plugins.shade.resource
    .ApacheLicenseResourceTransformer">
        </transformer>
        <transformer implementation=
    "org.apache.maven.plugins.shade
    .resource.AppendingTransformer">
          <resource>reference.conf</resource>
        </transformer>
```

```
      </transformers>
    </configuration>
    <executions>
      <execution>
        <phase>package</phase>
        <goals>
          <goal>shade</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

Then run `mvn assembly` and the resulting `jar` file can be run as the preceding code, but leaving out the Spark assembly `jar` file from the classpath.

Some of the following links useful for referencing are as follows:

- `http://maven.apache.org/guides/getting-started/`
- `http://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html`
- `http://maven.apache.org/plugins/maven-dependency-plugin/`

Building your Spark job with something else

If neither sbt nor Maven suits your needs, you may decide to use another build system. Thankfully, Spark supports building a fat JAR file with all the dependencies of Spark, which makes it easy to include in the build system of your choice. Simply run `sbt/sbt assembly` in the Spark directory and copy the resulting assembly JAR file from `core/target/spark-core-assembly-0.7.0.jar` to your build dependencies, and you are good to go.

No matter what your build system is, you may find yourself wanting to use a patched version of the Spark libraries. In that case, you can deploy your Spark library locally. I recommend giving it a different version number to ensure that `sbt/maven` picks up the modified version. You can change the version by editing `project/SparkBuild.scala` and changing the `version :=` part of the code. If you are using sbt, you should run an `sbt/sbt update` in the project that is importing the custom version. For other build systems, you just need to ensure that you use the new assembly `jar` file as part of your build.

Summary

So, now you can build your Spark jobs with Maven, sbt, or a build system of your choice. It's time to jump in and start learning how to do more fun and exciting things, such as how to create a Spark context, in the subsequent chapter.

4
Creating a SparkContext

This chapter will cover how to create a `SparkContext` context for your cluster. A `SparkContext` class represents the connection to a Spark cluster and provides the entry point for interacting with Spark. We need to create a `SparkContext` instance so that we can interact with Spark and distribute our jobs. In *Chapter 2*, *Using the Spark Shell*, we interacted with Spark through the Spark shell, which created a `SparkContext`. Now you can create RDDs, broadcast variables, counters, and so on, and actually do fun things with your data. The Spark shell serves as an example of interaction with the Spark cluster through `SparkContext` in `./repl/src/main/scala/spark/repl/SparkILoop.scala`.

The following code snippet creates a `SparkContext` instance using the `MASTER` environment variable (or `local`, if none are set) called `Spark shell` and doesn't specify any dependencies. This is because the Spark shell is built into Spark and, as such, doesn't have any JAR files that it needs to be distributed.

```
def createSparkContext(): SparkContext = {
  val master = this.master match {
    case Some(m) => m
    case None => {
     val prop = System.getenv("MASTER")
     if (prop != null) prop else "local"
    }
  }
  sparkContext = new SparkContext(master, "Spark shell")
  sparkContext
  }
```

For a client to establish a connection to the Spark cluster, the `SparkContext` object needs some basic information as follows:

- `master`: The `master` URL can be in one of the following formats:
  - `local[n]`: for a local mode
  - `spark://[sparkip]`: to point to a Spark cluster
  - `mesos://`: for a mesos path if you are running a mesos cluster
- `application name`: This is the human-readable application name
- `sparkHome`: This is the path to Spark on the master/workers machines
- `jars`: This gives the path to the list of JAR files required for your job

Scala

In a Scala program, you can create a `SparkContext` instance using the following code:

```
val spar kContext = new SparkContext(master_path, "application
name", ["optional spark home path"],["optional list of jars"])
```

While you can hardcode all of these values, it's better to read them from the environment with reasonable defaults. This approach provides maximum flexibility to run the code in a changing environment without having to recompile the code. Using `local` as the default value for the master machine makes it easy to launch your application locally in a test environment. By carefully selecting the defaults, you can avoid having to over-specify them. An example would be as follows:

```
import spark.sparkContext
import spark.sparkContext._
import scala.util.Properties

val master = Properties.envOrElse("MASTER","local")
val sparkHome = Properties.get("SPARK_HOME")
val myJars = Seq(System.get("JARS")
val sparkContext = new SparkContext(master, "my app", sparkHome,
myJars)
```

Java

To create a `SparkContext` instance in Java, try the following code:

```
import spark.api.java.JavaSparkContext;

JavaSparkContext ctx = new JavaSparkContext("master_url",
"application name", ["path_to_spark_home", "path_to_jars"]);
```

While the preceding code works (once you have replaced the parameters with the correct values for your setup), it requires a code change if you've changed any of the parameters. Instead, use reasonable defaults and allow them to be overridden similar to the example Scala code. The following illustrates how to do this with the environment variables:

```
String master = System.getEnv("MASTER");
if (master == null) {
  master = "local";
}
String sparkHome = System.getEnv("SPARK_HOME");
if (sparkHome == null) {
  sparkHome = "./";
}
String jars = System.getEnv("JARS");
JavaSparkContext ctx = new JavaSparkContext(System.getenv("MASTER"),
"my Java app",
System.getenv("SPARK_HOME"), System.getenv("JARS"));
```

Shared Java and Scala APIs

Once you have a `SparkContext` created, it will serve as your main entry point. In the next chapter, you will learn how to use our `SparkContext` instance to load and save data. You can also use the `SparkContext` instance to launch more Spark jobs and add or remove dependencies. Some of the non-data-driven methods you can use on the `SparkContext` instance are as follows:

Method	Use
`addJar(path)`	Adds the JAR file for all future jobs run through the `SparkContext` instance
`addFile(path)`	Downloads the file to all nodes on the cluster
`stop()`	Shuts down the `SparkContext` connection
`clearFiles()`	Removes the files so that new nodes will not download them
`clearJars()`	Removes the JAR files from being required for future jobs

Python

The Python `SparkContext` is a bit different from the Scala and Java contexts since Python doesn't use JAR files to distribute dependencies. Since you are still likely to have dependencies, set `pyFiles` with the ZIP and PY files as desired on SparkContext (or leave it empty if you don't have any files to distribute).

You can create a Python `SparkContext` using the following code:

```
from pyspark import SparkContext

sc = SparkContext("master","my python app", sparkHome="sparkhome",
pyFiles="placeholderdeps.zip")
```

Now you are able to create a connection to your Spark cluster, so it's time to get started on loading our data into Spark.

Links and references

Some useful links for referencing are listed as follows:

- `http://spark-project.org/docs/latest/quick-start.html`
- `http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html`
- `https://github.com/mesos/spark/blob/master/repl/src/main/scala/spark/repl/SparkILoop.scala`
- `http://spark-project.org/docs/0.7.0/api/pyspark/pyspark.context.SparkContext-class.html`
- `http://spark-project.org/docs/0.7.0/api/core/spark/SparkContext.html`
- `http://spark-project.org/docs/0.7.0/api/core/spark/api/java/JavaSparkContext.html`
- `http://www.scala-lang.org/api/current/index.html#scala.util.Properties$`

Summary

In this chapter, we've covered how to connect to our Spark cluster using `SparkContext`. Using `SparkContext`, we will start to look at the different data sources that we can use to load data into Spark in the next chapter.

5
Loading and Saving Data in Spark

By this point in the book you have experimented with the Spark shell, figured out how to create a connection to the Spark cluster, and built jobs for deployment. Now to make those jobs useful, you will learn how to load and save data in Spark. Spark's primary unit for data representation is an RDD, which allows for easy parallel operations on the data. Other forms of data, such as counters, have their own representation. Spark can load and save RDDs from a variety of sources.

RDDs

Spark RDDs can be created from any supported Hadoop source. Native collections in Scala, Java, and Python can also serve as the basis for an RDD. Creating RDDs from a native collection is especially useful for testing.

Before jumping into the details on the supported data sources/sinks, take some time to learn about what RDDs are and what they are not. It is crucial to understand that even though an RDD is defined, it does not actually contain data. This means that when you go to access the data in an RDD it could fail. The computation to create the data in an RDD is only done when the data is referenced; for example, it is created by caching or writing out the RDD. This means that you can chain a large number of operations together, and not have to worry about excessive blocking. It's important to note that during the application development, you can write code, compile it, and even run your job, and unless you materialize the RDD, your code may not have even tried to load the original data.

Each time you materialize an RDD it is re-computed. If we are going to be using something frequently, a performance improvement can be achieved by caching the RDD.

Loading data into an RDD

Now the chapter will examine the different sources you can use for your RDD. If you decide to run through the examples in the Spark shell, you can call `.cache()` or `.first()` on the RDDs you generate to verify that it can be loaded. In *Chapter 2*, *Using the Spark Shell*, you learned how to load data text from a file and from the S3 storage system, where you can look at different formats of data and the different sources that are supported.

One of the easiest ways of creating an RDD is taking an existing Scala collection and converting it into an RDD. The Spark context provides a function called `parallelize`; this takes a Scala collection and turns it into an RDD that is of the same type as the data input.

- **Scala**:

```
val dataRDD = sc.parallelize(List(1,2,4))
```

- **Java**:

```
JavaRDD<Integer> dataRDD = sc.parallelize(Arrays.asList(1,2,4));
```

- **Python**:

```
rdd = sc.parallelize([1,2,3])
```

The simplest method for loading external data is loading text from a file. This requires the file to be available on all the nodes in the cluster, which isn't much of a problem for a local mode. When in a distributed mode, you will want to use Spark's `addFile` functionality to copy the file to all the machines in your cluster. Assuming your `SparkContext` is called `sc`, we could load text data from a file (you need to create the file):

- **Scala**:

```
import spark.SparkFiles;
...
sc.addFile("spam.data")
val inFile = sc.textFile(SparkFiles.get("spam.data"))
```

- **Java**:

```
import spark.Sparkfiles;

sc.addFile("spam.data");
JavaRDD<String> lines = sc.textFile(SparkFiles.get("spam.data"));
```

- **Python**:

```
from pyspark.files import SparkFiles

sc.addFile("spam.data")
sc.textFile(SparkFiles.get("spam.data"))
```

The resulting RDD is an overridden string with each line being a unique element in the RDD. Frequently, your input files will be CSV or TSV files, which you will want to parse using one of the standard CSV libraries. In *Chapter 2*, *Using the Spark Shell*, you parsed them with a split and `toDouble`, but that doesn't always work out so well for more complex CSV files. Looking back to *Chapter 3*, *Building and Running a Spark Application* where you learned how to build jobs, you can change the `libraryDependencies` in `build.sbt` to be:

```
libraryDependencies ++= Seq(
    "org.spark-project" % "spark-core_2.9.2" % "0.7.0",
    "net.sf.opencsv" % "opencsv" % "2.0"
)
```

This brings in a CSV parser to use. This chapter uses `opencsv` for this example for brevity's sake, but you may find another CSV parser better suited to your needs depending on what you are parsing. Let's look at a sample that parses the input CSV and sums all the rows:

```
package pandaspark.examples

import spark.SparkContext
import spark.SparkContext._
import spark.SparkFiles;
import au.com.bytecode.opencsv.CSVReader
import java.io.StringReader

object LoadCsvExample {
  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: LoadCsvExample <master>
    <inputfile>")
      System.exit(1)
    }
```

```
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV Example",
                System.getenv("SPARK_HOME"),
                Seq(System.getenv("JARS")))
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val splitLines = inFile.map(line => {
      val reader = new CSVReader(new StringReader(line))
      reader.readNext()
    })
    val numericData = splitLines.map(line => line.map(_.toDouble))
    val summedData = numericData.map(row => row.sum)
    println(summedData.collect().mkString(","))
  }

}
```

The previous code also illustrates one of the ways of getting the data out of Spark: you can transform it to a standard Scala array using the `collect()` function. The `collect()` function is especially useful for testing, in much the same way as the `parallelize()` function is. The `collect()` function only works if your data fits in memory on a single host; in that case it adds the bottleneck of everything having to come back to a single machine.

While loading text files into Spark is certainly easy, text files on a local disk are often not the most convenient format for storing large chunks of data. Spark supports loading from all the different Hadoop formats (sequence files, regular text files, and so on) and from all the support Hadoop storage sources (HDFS, S3, HBase, and so on). If you want you can also load your CSV into HBase using some of its bulk loading tools (such as ImportTsv) and get at your CSV data that way. As of Version 0.7, PySpark does not support any of the advanced methods of loading data we will be discussing in the rest of this chapter.

Sequence files are binary-flat files consisting of key-value pairs, and they are one of the common ways of storing data for use with Hadoop. Loading a sequence file into Spark is similar to loading a text file, but you also need to let it know about the types of the keys and values. The types must either be subclasses of Hadoop's `Writable` class or be implicitly convertible to such a type. For Scala users, some natives are convertible through implicit conversions in `WritableConverter`. As of Version 0.7, the standard `WritableConverter` types are `Int`, `Long`, `Double`, `Float`, `Boolean`, `Array`, and `String`.

Let's illustrate this by looking at how to load a sequence file of `String` to `Integer`.

- **Scala**:

```
val data = sc.sequenceFile[String, Int](inputFile)
```

- **Java**:

```
JavaPairRDD<Text, IntWritable> dataRDD = sc.sequenceFile(file,
Text.class, IntWritable.class);
JavaPairRDD<String, Integer> cleanData = dataRDD.map(new
PairFunction<Tuple2<Text, IntWritable>, String, Integer>() {
 @Override
public Tuple2<String, Integer> call(Tuple2<Text, IntWritable>
pair) {
return new Tuple2<String, Integer>(pair._1().toString(),
pair._2().get());
}
});
```

Note that in the preceding cases, like with the text input, the file need not be a traditional file; it can reside on S3, HDFS, and so on. Also note that for Java, you can't rely on implicit conversions between types.

HBase is a Hadoop-based database designed to support random read/write access to entries. Loading data from HBase is a bit different from text files and sequence files. With HBase, we have to specify the type information of our data to Spark in a different way. Since HBase isn't included by default as a dependency of Spark, you will need to add it to your build system like you did with `opencsv` previously by adding `org.apache.hbase" % "hbase" % "0.94.6`.

If you run into difficulty with unresolved dependencies, make sure to add the Apache HBase release Maven repository at `https://repository.apache.org/content/repositories/releases` to your resolvers.

Let's illustrate the use of HBase database:

- **Scala**:

```
import spark._
import org.apache.hadoop.hbase.{HBaseConfiguration,
HTableDescriptor}
import org.apache.hadoop.hbase.client.HBaseAdmin
```

```
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
…
val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, input_table)
 // Initialize hBase table if necessary
val admin = new HBaseAdmin(conf)
if(!admin.isTableAvailable(input_table)) {
  val tableDesc = new HTableDescriptor(input_table)
  admin.createTable(tableDesc)
}
val hBaseRDD = sc.newAPIHadoopRDD(conf,
                  classOf[TableInputFormat],
                  classOf[org.apache.hadoop.
                  hbase.io.ImmutableBytesWritable],
                  classOf[org.apache.hadoop.
                  hbase.client.Result])
```

- **Java**:

```
import spark.api.java.JavaPairRDD;
import spark.api.java.JavaSparkContext;
import spark.api.java.function.FlatMapFunction;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.client.HBaseAdmin;
import org.apache.hadoop.hbase.mapreduce.TableInputFormat;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.hbase.client.Result;
...
JavaSparkContext sc = new JavaSparkContext(args[0], "sequence
load", System.getenv("SPARK_HOME"), System.getenv("JARS"));
Configuration conf = HBaseConfiguration.create();
conf.set(TableInputFormat.INPUT_TABLE, args[1]);
//Initialize HBase table if necessary
HBaseAdmin admin = new HBaseAdmin(conf);
if(!admin.isTableAvailable(args[1])) {
    HTableDescriptor tableDesc = new
       HTableDescriptor(args[1]);
    admin.createTable(tableDesc);
}
JavaPairRDD<ImmutableBytesWritable, Result> hBaseRDD
= sc.newAPIHadoopRDD( conf, TableInputFormat.class,
ImmutableBytesWritable.class, Result.class);
```

The method that you used to load the HBase data can be generalized for loading all other sorts of Hadoop data. If a helper method in Spark context does not already exist for loading the data, simply create a configuration specifying how to load the data and pass it into the `newAPIHadoopRDD` method. Helper methods exist for plain text files and sequence files. A helper method also exists for Hadoop files similar to the Sequence File API.

Saving your data

While distributed computational jobs are a lot of fun, they are much more applicable when the results get stored somewhere useful. While the methods for loading an RDD are largely found in the `SparkContext` class, the methods for saving an RDD are defined on the RDD classes. In Scala, implicit conversion exists so that an RDD that can be saved as a sequence file is converted to the appropriate type, and in Java explicit conversion must be used.

Here are the different ways to save an RDD:

- **Scala**:

```
rddOfStrings.saveAsTextFile("out.txt")
keyValueRdd.saveAsSequenceFile("sequenceOut")
```

- **Java**:

```
rddOfStrings.saveAsTextFile("out.txt")
keyValueRdd.saveAsSequenceFile("sequenceOut")
```

- **Python**:

```
rddOfStrings.saveAsTextFile("out.txt")
```

Links and references

Some of the useful links that you can use for references are as follows:

- `http://spark-project.org/docs/latest/scala-programming-guide.html#hadoop-datasets`
- `http://opencsv.sourceforge.net/`
- `http://commons.apache.org/proper/commons-csv/`
- `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/SequenceFileInputFormat.html`

- `http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/InputFormat.html`
- `http://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-single-node-cluster/`
- `http://spark-project.org/docs/latest/api/pyspark/index.html`
- `http://wiki.apache.org/hadoop/SequenceFile`
- `http://hbase.apache.org/book/quickstart.html`
- `http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/mapreduce/TableInputFormat.html`
- `http://spark-project.org/docs/latest/api/core/index.html#spark.api.java.JavaPairRDD`

Summary

In this chapter, we have seen how to load data from a variety of different sources. We have also looked at basic parsing of the data from text input files. Now that we can get our data loaded into a Spark RDD, it is time to explore the different operations we can perform on our data in the next chapter.

6
Manipulating Your RDD

The last few chapters have been the necessary groundwork for getting Spark working. Now that you know how to load and save your data in different ways, it's time for the big payoff: manipulating the data. The API for manipulating your RDD is similar between the languages, but not identical. Unlike the previous chapters, each language is covered in its own section; you probably only need to read the one pertaining to the language you are interested in using. Particularly, the Python implementation is currently not on feature parity with the Scala/Java API, but it supports most of the basic functionalities as of 0.7 with plans for future versions to improve feature parity.

Manipulating your RDD in Scala and Java

Manipulating your RDD in Scala is quite simple, especially if you are familiar with Scala's collection library. Many of the standard functional list functions are available directly on Spark's RDDs with the primary catch being that one can't rely on them being executed on the same machine. This makes porting of the existing Scala code to be distributed in a much simpler way than porting of the, say, Java or Python code.

Manipulating your RDD in Java is fairly simple, but a little more awkward at times than in Scala. As Java doesn't have implicit conversions, we have to be more explicit with our types. While the return types are Java friendly, Spark requires the use of Scala's `Tuple2` class for key-value pairs.

The hallmark of a MapReduce system are the two commands: `map` and `reduce`. We've seen the `map` function used in the previous chapters. The `map` function works by taking in a function that works on each individual element in the input RDD and produces a new output element. For example, to produce a new RDD where you add one to every number, use `rdd.map(x => x+1)` or in Java, you can use the following code:

```
rdd.map(new Function<Integer, Integer>() {
  public Integer call(Integer x) { return x+1;}
});
```

It is important to understand that the `map` function and the other Spark functions do not transform the existing elements, rather they return a new RDD with the new elements. The `reduce` function takes a function that operates on pairs to combine all the data. The function you provide needs to be commutative and associative (that is, f(a,b) == f(b,a) and f(a,f(b,c)) == f(f(a,b),c). For example, to sum all the elements, use `rdd.reduce(x,y => x+y)` or `rdd.reduce(new Function2<Integer, Integer, Integer>(){ public Integer call(Integer x, Integer y) { return x+y;} }`.

The `flatMap` function is a useful utility, which lets you write a function that returns an `Iterable` object of the type you want and then flattens the results. A simple example of this is a case where you want to parse all the data, but may fail to parse some of it. The `flatMap` function can be used to output an empty list if it failed, or a list with the success if it worked. In addition to the `reduce` function, there is a corresponding `reduceByKey` function that works on RDDs, which are key-value pairs to produce another RDD. Unlike when using map on a list in Scala, your function will run on a number of different machines, so you can't depend on a shared state with this.

Before continuing with the other wonderful functions for manipulating your RDD, first you need to read a bit about shared states. In the preceding example where we added one to every integer, we didn't really share states. However, for even simple tasks, like the distributed parsing of data that we did when loading the CSV file, it can be quite handy to have shared counters for things like keeping track of the number of rejected records. Spark supports both shared immutable data, which it calls `broadcast` and `accumulator` variables. You can create a new broadcast by calling `sc.broadcast(value)`. While you don't have to explicitly broadcast values as Spark does its magic in the background, broadcasting ensures that the value is sent to each node only once. The `broadcast` variables are often used for operations such as side inputs (for example, a hashmap), which need to look up as part of the `map` function. This returns an object that can be used to reference the `broadcast` value.

Another method of sharing state is with an `accumulator` variable. To create an `accumulator` variable, use `sc.accumulator(initialvalue)`. This returns an object that you can add to in a distributed context and then get back the value by calling `.value()`. The `accumulableCollection` function can be used to create a collection that is appended in a distributed fashion; however, if you find yourself using this, ask yourself if you could use the results of a map output better. If the predefined accumulators don't work for your use case, you can use `accumulable` to define your own accumulation type. A `broadcast` value can be read by all the workers and an `accumulator` value can be written to by all the workers and only read by the driver.

If you are writing Scala code that interacts with a Java Spark process (say for testing), you may find it useful to use `intAccumulator` and similar methods on the Java Spark context, otherwise your `accumulator` types might not quite match up.

If you find your `accumulator` variable isn't increasing in value like you expect, remember that Spark is lazy. This means that Spark won't actually perform the maps, reduces, or other computation on RDDs until the data outputs the computations.

Look at your previous example that parsed CSV files and made it a bit more robust. In your previous work, you assumed the input was well formatted and if any errors occurred, our entire pipeline would fail. While this can be the correct behavior for some work, when dealing with data from third-party parties, we may want to accept some number of malformed records. On the other hand, we don't want to just throw out all the records and declare it a success; we might miss an important format change and produce meaningless results. Let's add counters for errors to our code:

```
package spark.examples

import spark.SparkContext
import spark.SparkContext._
import spark.SparkFiles;

import au.com.bytecode.opencsv.CSVReader

import java.io.StringReader

object LoadCsvWithCountersExample {
  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: LoadCsvExample <master>
                          <inputfile>")
      System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV With Counters
             Example",
  System.getenv("SPARK_HOME"),
  Seq(System.getenv("JARS")))
    val invalidLineCounter = sc.accumulator(0)
    val invalidNumericLineCounter = sc.accumulator(0)
    sc.addFile(inputFile)
```

```
    val inFile = sc.textFile(inputFile)
    val splitLines = inFile.flatMap(line => {
      try {
        val reader = new CSVReader(new StringReader(line))
        Some(reader.readNext())
      }
      catch {
        case _ => {
          invalidLineCounter += 1
          None
        }
      }
    }
    )
    val numericData = splitLines.flatMap(line => {
      try {
        Some(line.map(_.toDouble))
      }
      catch {
        case _ => {
          invalidNumericLineCounter += 1
          None
        }
      }
    }
    )
    val summedData = numericData.map(row => row.sum)
    println(summedData.collect().mkString(","))
    println("Errors:
            "+invalidLineCounter+","+invalidNumericLineCounter)
  }

}
```

Or in Java:

```
import spark.Accumulator;
import spark.api.java.JavaRDD;
import spark.api.java.JavaPairRDD;
import spark.api.java.JavaSparkContext;
import spark.api.java.function.FlatMapFunction;

import au.com.bytecode.opencsv.CSVReader;
```

```
import java.io.StringReader;
import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;

public class JavaLoadCsvCounters {
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: JavaLoadCsvCounters <master>
                         <inputfile>");
      System.exit(1);
    }
    String master = args[0];
    String inputFile = args[1];
    JavaSparkContext sc = new JavaSparkContext(master,
                          "java load csv with counters",
                          System.getenv("SPARK_HOME"),
                          System.getenv("JARS"));
    final Accumulator<Integer> errors = sc.accumulator(0);
    JavaRDD<String> inFile = sc.textFile(inputFile);
    JavaRDD<Integer[] > splitLines = inFile.flatMap(
                                     new FlatMapFunction<String,
                                     Integer[]> (){
      public Iterable<Integer[]> call(String line) {
        ArrayList<Integer[]> result = new ArrayList<Integer[]>();
        try {
               CSVReader reader = new CSVReader(
                                  new StringReader(line));
               String[] parsedLine = reader.readNext();
               Integer[] intLine = new Integer[parsedLine.length];
               for (int i = 0; i < parsedLine.length; i++) {
                 intLine[i] = Integer.parseInt(parsedLine[i]);
               }
               result.add(intLine);
        }
        catch (Exception e) {
          errors.add(1);
        }
        return result;
        }
  }
  );
    System.out.println("Loaded data "+splitLines.collect());
    System.out.println("Error count "+errors.value());
  }
}
```

The preceding code example illustrates the use of the `flatMap` method. In general, `flatMap` can be used when your map function returns a sequence of the type you are returning and flattens it. Since options in Scala can be used as sequences through an implicit conversion, you can avoid having to explicitly filter out the `None` result and just use `flatMap`.

Summary statistics can be quite useful when examining large data sets. In the preceding example, you loaded the data as `Double`, to use Spark's provided summary statistics capabilities on the RDD. In Java, this requires explicit use of the `JavaDoubleRDD` type. It is also important to use `DoubleFunction<Integer[]>` rather than `Function<Integer[], Double>` in the following example, since the first won't result in the type `JavaDoubleRDD`. No such consideration is required for Scala as implicit conversions deal with the details. Compute the mean and the variance, or compute them together with the statistics. You can extend this by adding on the end of the preceding function to print out the summary statistics as `println(summedData.stats())`:

To do this in Java, you can add the following code:

```
JavaDoubleRDD summedData = splitLines.map(
                           new DoubleFunction<Integer[]>()
{
  public Double call(Integer[] in) {
    Double ret = 0.;
    for (int i = 0; i < in.length; i++) {
      ret += in[i];
    }
    return ret;
  }
});
System.out.println(summedData.stats());
```

When working with key-value pair data, it can be quite useful to group data with the same key together (for example, if the key represents the user or sample). The `groupByKey` function provides an easy way to group data together by keys. The `groupByKey` key-value pair is a special case of `combineByKey`. There are several functions in the `PairRDD` class, which are all implemented very closely on top of `combineByKey`. If you find yourself using `groupByKey` or one of the other functions derived from `combineByKey` and immediately transforming the result, you should check if the function is better suited to the task. A common thing to do while starting out is to use `groupByKey` and then sum the results with `groupByKey().map((x,y) => (x,y.sum))`, or you can also use the following code in Java:

```
pairData.groupByKey().mapValues(new Function<List<Integer>,
  Integer >(){
```

```
    public Integer call(List<Integer> x){
      Integer sum = 0;
      for (Integer i : x) {
        sum += i;
      }
      return sum;
    }
  });
```

By using `reduceByKey`, it could be simplified to `reduceByKey((x,y) => x+y)`, or to do this in Java you can use the following code:

```
pairData.reduceByKey(new Function2<Integer, Integer, Integer>() {
  public Integer call (Integer a, Integer b){
    return a+b;
  }
});
```

The `foldByKey(zeroValue) (function)` method is similar to a traditional `fold` operation that works on per key. On a list in a traditional `fold` function, the provided values would be called with the initial value and the first element of the list, and then the resulting value and the next element of the list would be the input to the next call of `fold`. Doing this requires sequentially processing the entire list, so `foldByKey` behaves slightly differently. There is a handy table of functions of `PairRDD` at the end of this section. Some of the `PairRDD` functionality was only added to the Java API in 0.7.2.

Sometimes, you will only want to update the values of a key-value pair data structure, such as a `PairRDD`. You've learned about `foldByKey` and how it doesn't quite work as a traditional `fold`. For Scala developers, if you require the "traditional" `fold` behavior, you can do a `groupByKey` and then map a fold over the resulting RDD by value. This is an example of a case where you only want to change the value and don't care about the key of the RDD, so examine the following code:

```
rdd.groupByKey().mapValues(x => {x.fold(0)((a,b) => a+b)})
```

Often your data won't be a clean value from a single source and you will want to join the data together for processing, which can be done with `coGroup`. This can be done when you are joining web access logs with transaction data or even just joining two different computations on the same data. Provided that the RDDs have the same key, we can join two RDDs together with `rdd.coGroup(otherRdd)`. There are a number of different join functions for different purposes, illustrated in the table at the end of this section.

The next task you will learn is distributing files among the clusters. We illustrate this by adding GeoIP support and mixing it together with the gradient descent example from the earlier chapter. Sometimes the libraries you use need files distributed along with them. While it is possible to add them to JAR and access them as class objects, Spark provides a simple way to distribute the required files by calling `addFile()` as shown in the following code:

```
package pandaspark.examples
import scala.math

import spark.SparkContext
import spark.SparkContext._
import spark.SparkFiles;
import spark.util.Vector

import au.com.bytecode.opencsv.CSVReader

import java.util.Random
import java.io.StringReader
import java.io.File

import com.snowplowanalytics.maxmind.geoip.IpGeo

case class DataPoint(x: Vector, y: Double)

object GeoIpExample {

    def main(args: Array[String]) {
      if (args.length != 2) {
        System.err.println("Usage: GeoIpExample <master>
                           <inputfile>")
        System.exit(1)
    }
    val master = args(0)
    val inputFile = args(1)
    val iterations = 100
    val maxMindPath = "GeoLiteCity.dat"
    val sc = new SparkContext(master, "GeoIpExample",
                System.getenv("SPARK_HOME"),
                Seq(System.getenv("JARS")))
    val invalidLineCounter = sc.accumulator(0)
    val inFile = sc.textFile(inputFile)
    val parsedInput = inFile.flatMap(line => {
```

```
        try {
              val row = (new CSVReader(
                new StringReader(line))).readNext()
                    Some((row(0),row.drop(1).map(_.toDouble)))
        }
        catch {
          case _ => {
            invalidLineCounter += 1
            None
          }
        }
      })
      val geoFile = sc.addFile(maxMindPath)
      //getLocation gives back an option so we use flatMap to only
  output if it's a some type
      val ipCountries = parsedInput.flatMapWith(_ => IpGeo
       (dbFile = SparkFiles.get(maxMindPath)))
       ((pair, ipGeo) => {
       ipGeo.getLocation(pair._1).map(c => (pair._1,
        c.countryCode)).toSeq
       })
      ipCountries.cache()
      val countries = ipCountries.values.distinct().collect()
      val countriesBc = sc.broadcast(countries)
      val countriesSignal = ipCountries.mapValues(
        country => countriesBc.value.map(
        s => if (country == s) 1
             else 0
      ))
      val dataPoints = parsedInput.join(countriesSignal)
         .map(input => {
            input._2 match {
              case (countryData, originalData) => DataPoint(
                new Vector(countryData++originalData
                .slice(1,originalData.size-2)),
                originalData(originalData.size-1))
             }
         })
      countriesSignal.cache()
      dataPoints.cache()
      val rand = new Random(53)
      var w = Vector(dataPoints.first.x.length, _ =>
        rand.nextDouble)
      for (i <- 1 to iterations) {
        val gradient = dataPoints.map(p =>
          (1 / (1 + math.exp(-p.y*(w dot p.x))) - 1) * p.y *
          p.x).reduce(_ + _)
```

```
            w -= gradient
        }
        println("Final w: "+w)
      }
    }
```

There have been issues with `addFile` in the local mode in the past. If you run into this, you can try the workaround that the Shark (a tool we cover later) developers used (essentially using the result from `addFile` if it exists and falling back on the original file if it doesn't).

To know more about this, you can refer to:

`https://github.com/amplab/shark/commit/47c21f55621acd5afb412f54a45c68e141240030`.

In the preceding code, you see multiple Spark computations. The first is to determine all the countries that our data covers, so we can map each country to a binary feature. The code then uses a public list of proxies and the reported latency to try to estimate the latency I measured. This also illustrates the use of `mapWith`. If you have a mapping job that needs to create a per partition resource, `mapWith` can be used to do this. This can be useful for connections to backends or the creation of something like a **PRNG** (**pseudorandom number generator**). Some elements also can't be serialized over the wire (such as the IpCountry, in the example), so you have to create them per share. You can also see that we cache a number of our RDDs to keep them from having to be re-computed.

There are several options when working with multiple RDDs.

Scala RDD functions

`PairRDD` functions are based on `combineByKey`. All operate on RDDs of type `[K,V]` as shown in the following table:

Function	Parameter options	Explanation	Return type

`foldByKey`	`(zeroValue)(func(V,V)=>V)`
	`(zeroValue, partitioner)(func(V,V=>V)`
	`(zeroValue, partitions)(func(V,V=>V)`
		Merges the values using the provided function. Unlike a traditional `fold` function over a list, the `zeroValue` can be added an arbitrary number of times.	`RDD[K,V]`

Function	Parameter options	Explanation	Return type
`reduceByKey`	`(func(V,V)=>V)` `(func(V,V)=>V, numTasks)`	Parallel version of `reduce` that merges the values for each key using the provided function and returns an RDD.	`RDD[K,V]`
`groupByKey`	`()` `(numPartitions)`	Groups elements together by key.	`RDD[K,Seq[V]]`

Functions for joining PairRDD functions

Often when working with two or more key-value RDDs, it is useful to join them together. There are a few different methods to do this depending on what your desired behavior is, as shown in the following table:

Function	Parameter options	Explanation	Return type
`cogroup`	`(otherRDD[K,W]...)`	Joins two (or more) RDDs by the shared key. Note that if an element is missing in one RDD but present in the other, one of the Seq will simply be empty.	`RDD[(K,(Seq[V], Seq[W]...))]`

`join`	`(otherRDD[K,W])`
	`(otherRDD[K,W], partitioner)`
	`(otherRDD[K,W], numPartitions)`
		Joins an RDD with another RDD. The result is only present for elements where the key is present in both RDDs.	`RDD[(K,(V,W))]`

`subtractKey`	`(otherRDD[K,W])`
	`(otherRDD[K,W], partitioner)`
	`(otherRDD[K,W], numPartitions)`
		Returns an RDD with only keys not present in the other RDD.	`RDD[(K,V)]`

Other PairRDD functions

Some functions only make sense when working on key-value pairs.

Function	Parameter options	Explanation	Return type
`lookup`	`(key: K)`	Looks up a specific element in the RDD. Uses the RDD's partitioner to figure out which partition(s) to look at.	`Seq[V]`
`mapValues`	`(f: V => U)`	A specialized version of map for PairRDD when you only want to change the value of the key-value pair. This takes the provided `Map` function and applies it to the value. If you need to make your change based on both key and value, you must use one of the normal RDD `Map` functions.	`RDD[(K,U)]`
`collectAsMap`	`()`	Takes an RDD and returns a concrete map. Your RDD must be able to fit into the memory.	`Map[K, V]`
`countByKey`	`()`	Counts the number of elements for each key.	`Map[K, Long]`

Function	Parameter options	Explanation	Return type
`partitionBy`	`(partitioner: Partitioner, mapSideCombine : Boolean)`	Returns a new RDD with the same data but partitioned by the new `Partitioner`, and `mapSideCombine` controls Spark group values with the same key together before repartitioning. Defaults to false; set to true if you have a large percent of duplicate keys.	`RDD[(K,V)]`
`flatMapValues`	`(f: V => TraversableOnce[U] )`	Similar to `mapValues`. A specialized version of `flatMap` for PairRDDs when you only want to change the value of the key-value pair. Takes the provided Map function and applies it to the value. The resulting sequence is then "flattened"; that is, instead of getting `Seq[Seq[V]]`, you get `Seq[V]`. If you need to make your change based on both key and value, you must use one of the normal RDD map functions.	`RDD[(K,U)]`

For information on saving `PairRDD`, refer to the previous chapter.

DoubleRDD functions

Spark defines a number of convenience functions, which work when your RDD is comprised of `double` data types.

Function	Arguments	Returns
`mean`	`()`	Average of RDDs elements
`sampleStdev`	`()`	Standard deviation for a sample rather than a population (divides by N-1 rather than N)
`Stats`	`()`	Mean, variance, and count as a StatCounter
`Stdev`	`()`	Standard deviation (for population)
`Sum`	`()`	Sum of the elements
`variance`	`()`	Variance of RDDs elements

General RDD functions

The remaining RDD functions are defined on all RDDs.

Function	Arguments	Returns
`aggregate`	`(zeroValue: U)(seqOp: (U,T) => T, combOp (U, U) => U)`	Aggregates all the elements of each partition of an RDD, and then combines them using `combOp`. The argument `zeroValue` should be neutral (that is, `0` for `+` and `1` for `*`).
`cache`	`()`	Caches an RDD reused without recomputing. Same as `persist(StorageLevel.MEMORY_ONLY)`.
`collect`	`()`	An array of all the elements in the RDD.
`count`	`()`	The number of elements in an RDD.
`countByValue`	`()`	A map of value to the number of times that value occurs.
`distinct`	`()` `(partitions: Int)`	RDD containing only distinct elements.

Function	Arguments	Returns
filter	(f: T => Boolean)	RDD containing only elements matching f.
filterWith	(construct A: Int => A)(f: (T, A) => Boolean)	Similar to filter, but f takes an additional parameter generated by constructA, which is called per partition. The original motivation for this came from providing PRNG generation per partition.
first	()	The "first" element of the RDD.
flatMap	(f: T => TraversableOnce[U])	An RDD of type U.
fold	(zeroValue: T)(op: (T,T) => T)	Merges values using the provided operation, first on each partition, and then merges the merged result.
foreach	(f: T => Unit)	Applies the function f to each element.
groupBy	(f: T => K) (f: T => K, p: Partitioner) (f: T => K, numPartitions:Int)	Takes in an RDD and produces an RDD pair of type (K,Seq[T]) using the result of f for the key of each element.
keyBy	(f: T => K) (f: T => K, p: Partitioner) (f: T => K, numPartitions:Int)	Same as groupBy, but does not group results together with duplicate keys. Returns an RDD of (K,T).
map	(f: T => U)	An RDD of the result of applying f to every element in the input RDD.
mapPartitions	(f: Iterator[T] => Iterator[U])	Similar to map, except the provided function takes and returns an Iterator and is applied to each partition.
mapPartitionsWithIndex	(f: (Int, Iterator[T]) => Iterator[U], preservePartitions)	Same as mapPartitions, but also provides the index of the original partition.

Function	Arguments	Returns
`mapWith`	`(constructA: Int => A)(f: (T, A) => U)`	Similar to `map`, but `f` takes an additional parameter generated by `constructorA`, which is called per partition. The original motivation for this came from providing PRNG generation per partition.
`persist`	`()` `(newLevel: StorageLevel)`	Sets the RDD storage level, which can cause the RDD to be stored after it is computed. Different `StorageLevels` can be seen in `StorageLevel.scala` (`NONE`, `DISK_ONLY`, `MEMORY_ONLY`, and `MEMORY_AND_DISK` are the common ones).
`pipe`	`(command: Seq[String])` `(command: Seq[String], env: Map[String, String])`	Takes an RDD and calls the specified command with the optional environment and pipes each element through the command. Results in an RDD of `String` type.
`sample`	`(withReplacement: Boolean, fraction: Double, seed: Int)`	RDD of that fraction.
`takeSample`	`(withReplacement: Boolean, num: Int, seed: Int)`	An array of the requested number of elements. This works by oversampling the RDD and then grabbing a subset.
`toDebugString`	`()`	A handy function that outputs the recursive `deps` of the RDD.
`union`	`(other: RDD[T])`	An RDD containing elements of both RDDs. Duplicates are not removed.
`unpersist`	`()`	Remove all the persistent blocks of the RDD from the memory/disk.
`zip`	`(other: RDD[U])`	Requires that the RDDs have the same number of partitions of the same size. Returns an RDD of key-value pairs `RDD[T,U]`.

Java RDD functions

Many of the Java RDD functions are quite similar to the Scala RDD functions, but the type signatures are somewhat different.

Spark Java function classes

For the Java RDD API, we need to extend one of the provided function classes when implementing our function. The following table shows some of the Spark Java functions:

Name	Parameters	Purpose
`Function<T,R>`	`R apply(T t)`	This function takes something of type `T` and returns something of type `R`. Commonly used for maps.
`DoubleFunction<T>`	`Double apply(T t)`	Same as `Function<T, Double>`, but the result of the map-like call returns a `JavaDoubleRDD` (for summary statistics).
`PairFunction<T, K, V>`	`Tuple2<K, V> apply(T t)`	This function results in a `JavaPairRDD`. If working on a `JavaPairRDD<A,B>`, let `T` be of type `Tuple2<A,B>`.
`FlatMapFunction<T, R>`	`Iterable<R> apply(T t)`	This function is for producing an RDD through a `flatMap` function.
`PairFlatMapFunction<T, K, V>`	`Iterable<Tuple2<K, V>> apply(T t)`	This function results in a `JavaPairRDD`. If working on a `JavaPairRDD<A,B>`, let `T` be of type `Tuple2<A,B>`.
`DoubleFlatMapFunction<T>`	`Iterable<Double> apply(T t)`	Same as `FlatMapFunction<T, Double>`, but the result of the map-like call returns a `JavaDoubleRDD` (for summary statistics).
`Function2<T1, T2, R>`	`R apply(T1 t1, T2 t2)`	This function is for taking two inputs and returning an output. Used by `fold` and similar functions.

Common Java RDD functions

The following table explains RDD functions that are available regardless of the type of RDD.

Name	Parameters	Purpose
`cache`	`()`	Persists an RDD in memory.
`coalesce`	`numPartitions: Int`	Returns a new RDD with `numPartitions` partitions.
`collect`	`()`	Returns the List representation of the entire RDD.
`count`	`()`	Number of elements.
`countByValue`	`()`	A map of each unique value to the number of times that a value shows up.
`distinct`	`()` `(Int numPartitions)`	An RDD consisting of all the distinct elements of the RDD optionally in the provided number of partitions.
`filter`	`(Function<T, Boolean> f)`	An RDD containing only elements for which `f` returns `true`.
`first`	`()`	The first element of the RDD.

`flatMap`	`(FlatMapFunction<T, U> f)`
	`(DoubleFlatMapFunction<T> f)`
	`(PairFlatMapFunction<T, K, V> f)`
		An RDD of the specified type (`U`, `Double`, and `Pair<K,V>` respectively).

`fold`	`(T zeroValue, Function2<T, T, T> f)`	The result `T` and each partition is folded individually with the `zeroValue` and then the results are folded.
`foreach`	`(VoidFunction<T> f)`	Applies the function to each element in the RDD.

Name	Parameters	Purpose
groupBy	(Function<T, K> f) (Function<T, K> f, Int numPartitions)	A JavaPairRDD of grouped elements.
map	(DoubleFunction<T> f) (PairFunction<T, K2, V2> f) (Function<T, U> f)	An RDD of the appropriate type for the input function (see the previous table) by calling the provided function on each element in the input RDD.
mapPartitions	(DoubleFunction<Iterator<T>> f) (PairFunction<Iterator<T>, K2, V2> f) (Function<Iterator<T>, U> f)	Similar to map, but the provided function is called per partition. This can be useful if you have some setup work that you need to do for each partition.
reduce	(Function2<T, T, T> f)	Uses the provided function to reduce all the elements.
sample	(Boolean withReplacement, Double fraction, Int seed)	A smaller RDD consisting of only the requested fraction of the data.

Methods for combining JavaPairRDD functions

There are a number of different functions we can use to combine RDDs as shown in the following table:

Name	Parameters	Purpose
subtract	(JavaRDD<T> other) (JavaRDD<T> other, Partitioner p) (JavaRDD<T> other, Int numPartitions)	Returns an RDD with only the elements initially present in the first RDD and not present in the other RDD.
union	(JavaRDD<T> other)	The union of the two RDDs.

Name	Parameters	Purpose
`zip`	`(JavaRDD<U> other)`	Returns an RDD of key-value pairs `RDD[T,U]`. **Important**: This function requires that the RDDs have the same number of partitions and size.

JavaPairRDD functions

The following table explains some functions that are only defined on key-value pair RDDs:

Name	Parameters	Purpose

`cogroup`	`(JavaPairRDD<K, W> other)`
	`(JavaPairRDD<K, W> other, Int numPartitions)`
	`(JavaPairRDD<K, W> other1, JavaPairRDD<K, W> other2)`
	`(JavaPairRDD<K, W> other1, JavaPairRDD<K, W> other2, Int numPartitions)`
		Joins two (or more) RDDs by the shared key. Note that if an element is missing in one RDD but present in the other, one of the list will simply be empty.

`combineByKey`	`(Function<V, C> createCombiner, Function2<C, V, C> mergeValue, Function2<C,C,C> mergeCombiners)`	Generic function to combine elements by keys. The argument `createCombiner` turns something of type `V` into something of type `C`, `mergeValue` adds a type `V` to a type `C`, and `mergeCombiners` is used to combine two `C` types into a single `C`.
`collectAsMap`	`()`	Returns a `map` of the key-value pairs.
`countByKey`	`()`	Returns a `map` of the key to the number of elements with that key.

Name	Parameters	Purpose
`flatMapValues`	`(Function[T] f, Iterable[V] v)`	Returns an RDD of type `V`.
`join`	`(JavaPairRDD<K, W> other)` `(JavaPairRDD<K, W> other, Int integers)`	Joins an RDD with another RDD. The result is only present for elements where the key is present in both RDDs.
`keys`	`()`	Returns an RDD of only the keys.
`lookup`	`(Key k)`	Looks up a specific element in the RDD. Uses the RDDs' partitioner to figure out which partition(s) to look at.
`reduceByKey`	`(Function2[V,V,V] f)`	The `reduceByKey` is the parallel version of `reduce`, which merges the values for each key using the provided function and returns an RDD.

`sortByKey`	`(Comparator[K] comp, Boolean ascending)`
	`(Comparator[K] comp)`
	`(Boolean ascending)`
		Sorts the RDDs by keys, so each partition contains a fixed range.

`values`	`()`	Returns an RDD of only the values.

Manipulating your RDD in Python

Spark has a more limited API than Java and Scala, but supports most of the core functionalities.

The hallmarks of a MapReduce system are the two commands: `map` and `reduce`. You've seen the `map` function used in the past chapters. The `map` function works by taking in a function that works on each individual element in the input RDD and produces a new output element. For example, to produce a new RDD where you have added one to every number, you would use `rdd.map(lambda x: x+1)`. It's important to understand that the `map` function and the other Spark functions do not transform the existing elements, rather they return a new RDD with the new elements. The `reduce` function takes a function that operates on pairs to combine all the data. This is returned to the calling program. If you were to sum all the elements, you would use `rdd.reduce(lambda x, y: x+y)`.

The `flatMap` function is a useful utility that allows you to write a function which returns an `Iterable` object of the type you want and then flattens the results. A simple example of this is a case where you want to parse all the data, but some of the data may not be parsed. The `flatMap` function can output an empty list if it failed or a list with the success if it worked. In addition to `reduce`, there is a corresponding `reduceByKey` function that works on RDDs which are key-value pairs and produces another RDD.

Many of the mapping operations are also defined with a partition variant. In this case, the function you need to provide takes and returns an `Iterator` object that represents all the data on that partition. This can be quite useful if the operation you need to perform has to do extensive work on each partition, for example, establishing a connection to a backend server.

Often, your data can be expressed with key-value mappings. As such, many of the functions defined on the Python RDD class only work if your data is in a key-value mapping. The `mapValues` function is used when you only want to update the key-value pair you are working with.

Another variant on the traditional map function is `mapPartitions`, which works on a per-partition level. The primary reason for using `mapPartitions` is to create the setup for your map function, which can't be serialized across the network. A good example of this is creating an expensive connection to a backend service or parsing some expensive side input.

```
def f(iterator):
      //Expensive work goes here
     for i in iterator:
       yield per_element_function(i)
```

In addition to simple operations on the data, Spark provides support for `broadcast` and `accumulator` values. The `broadcast` values can be used to broadcast a read-only value to all the partitions that can save having to reserialize a given value multiple times. Accumulators allow all the partitions to add to the `accumulator` and the result can then be read on the master. You can create an accumulator by doing `counter = sc.accumulator(initialValue)`. If you want to add custom behavior, you can also provide an `AccumulatorParam` as an argument to the `accumulator` function. The return can then be incremented as `counter += x` on any of the workers. The resulting value can then be read with `counter.value()`. The `broadcast` value is created with `bc = sc.broadcast(value)` and then accessed by `bc.value()` by any worker. The `accumulator` value can only be read on the master and the `broadcast` value can be read on all the partitions.

Standard RDD functions

The following table explains some of the functions that are available on all RDDs in Python:

Name	Parameters	Purpose
`flatMap`	`(f, preservesPartitioning =False)`	Takes a function that returns an `Iterator` object of type `U` for each input of type `T` and returns a flattened RDD of type `U`.
`mapPartitions`	`(f, preservesPartitioning=False)`	Takes a function, and the function takes in an `Iterator` of type `T` and returns an Iterator of type `U` and results in an RDD of type `U`. So, for example, if we provided a function that took in an iterator of integers and returned an iterator of strings and called it on an RDD of integers we would get back an RDD of strings. Useful for map operations with expensive per machine setup work.
`filter`	`(f)`	Takes a function and returns an RDD with only the elements that the function returns `true` for.
`distinct`	`()`	Returns an RDD with distinct elements (for example, entering 1, 1, and 2 will output 1, 2).
`union`	`(other)`	Returns a union of two RDDs.
`cartesian`	`(other)`	Returns the cartesian product of the RDD with the other RDD.

Name	Parameters	Purpose
`groupBy`	`(f, numPartitions =None)`	Returns an RDD with the elements grouped together for the value that f outputs.
`pipe`	`(command, env={})`	Pipes each element of the RDD to the provided command and returns an RDD of the result.
`foreach`	`f`	Applies the function `f` to each element in the RDD.
`reduce`	`f`	Reduces the elements using the provided function.
`fold`	`zeroValue, op`	Each partition is folded individually with `zeroValue` and then the results are folded.
`countByValue`	`()`	Returns a dictionary mapping each distinct value to the number of times it is found in the RDD.
`take`	`num`	Returns a list of `num` elements. This can be slow for large values of `num`, so use `collect` if you want to get back the entire RDD.
`partitionBy`	`(numPartitions, partition Func = hash)`	Makes a new RDD partitioned by the provided partitioning function. The `partitionFunc` argument simply needs to map the input key to the integer space, at which point `partionBy` takes it mod `numPartitions`.

PairRDD functions

The following table explains some functions that are only available on key-value pair functions:

Name	Parameters	Purpose
`collectAsMap`	`()`	Returns a dictionary consisting of all the key-value pairs of the RDD.
`reduceByKey`	`(func, numPartitions=None)`	The `reduceByKey` function is the parallel version of `reduce`, which merges the values for each key using the provided function and returns an RDD.
`countByKey`	`()`	Returns a dictionary of the number of elements for each key.
`join`	`(other, numPartitions=None)`	Joins an RDD with another RDD. The result is only present for elements where the key is present in both RDDs. The value that gets stored for each key is a tuple of the values from each RDD.
`rightOuterJoin`	`(other, numPartitions=None)`	Joins an RDD with another RDD. This function outputs a given key-value pair only if the key is present in the RDD being joined with. If the key is not present in the source RDD, the first value in the tuple will be `None`.
`leftOuterJoin`	`(other, numPartitions=None)`	Joins an RDD with another RDD. This function outputs a given key-value pair only if the key is present in the source RDD. If the key is not present in the other RDD, the second value in the tuple will be `None`.

Name	Parameters	Purpose
`combineByKey`	`(createCombiner, mergeValues, mergeCombiners)`	Combines elements by keys. This function takes an RDD of type `(K,V)` and returns an RDD of type `(K,C)`. The argument `createCombiner` turns something of type `V` into something of type `C`, `mergeValue` adds a `V` to a `C`, and `mergeCombiners` is used to combine two `C` types into a single `C`.
`groupByKey`	`(numPartitions=None)`	Groups the values in the RDD by the keys they have.
`cogroup`	`(other, numPartitions=None)`	Joins two (or more) RDDs by the shared key. Note that if an element is missing in one RDD but present in the other one, the list will simply be empty.

Links and references

Some of the useful links for referencing are as follows:

- `http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.List`
- `http://spark.incubator.apache.org/docs/latest/api/core/index.html#spark.api.java.JavaRDD`
- `http://spark.incubator.apache.org/docs/latest/api/core/index.html#spark.api.java.JavaPairRDD`
- `http://spark.incubator.apache.org/docs/latest/api/core/index.html#spark.SparkContext`
- `http://spark.incubator.apache.org/docs/latest/api/core/index.html#packa`

Summary

This chapter looked at how to perform computations on our data in a distributed fashion once loaded into an RDD. Combined with our knowledge of how to load and save RDDs, we can now write distributed programs using Spark. In the next chapter, we will look at how to use Spark with Hive.

7
Shark – Using Spark with Hive

This chapter will cover how to use Spark with Hive, and how to integrate Hive queries with a Spark program. This chapter isn't needed to understand any of the following chapters, so if you don't want to learn about Hive, skip ahead on to the next chapter.

The following topics are covered in this chapter:

- Uses of Hive/Shark
- How to install Shark
- Loading data into Shark
- Running Shark
- Using HiveQL queries inside of a Spark program

Why Hive/Shark?

Hive is a popular Hadoop project that (among other things) allows for adhoc queries of large datasets. The query language for Hive is called HiveQL, and supports much of SQL as well as number of extensions. Shark is designed to be compatible with the Hive query language, serialization formats, and so on. People primarily choose to use Shark because it is much faster than traditional Hive and Hadoop for multiple queries. This chapter will not be able to teach you Hive if you don't already know it, but rather it will look at integrating HiveQL into your Spark programs and how to set up Shark. That being said, HiveQL is very similar to SQL, so if you have a strong grasp of SQL you can probably follow along reasonably well.

Installing Shark

As of the writing of this chapter, the latest version of Shark is v0.7.0 and it requires Spark 0.7.2 as well as a very recent JVM (Open JK7/Oracle HotSpot JDK7). Shark is available pre-built for both Hadoop 1 and Hadoop 2. As of the writing, the respective files are `http://spark-project.org/download/shark-0.7.0-hadoop1-bin.tgz` and `http://spark-project.org/download/shark-0.7.0-hadoop2-bin.tgz`. Once you have downloaded and extracted Shark, it's time to configure it. In this example, we will assume that you extracted in `/home/spark/`. Shark has a separate configuration from Spark, which lives at `shark-0.7.0/conf/shark-env.sh`. For local mode, you need to set up at least `HIVE_HOME` and `SPARK_HOME` like so:

```
export HIVE_HOME=/home/spark/hive-0.9.0-bin
export SPARK_HOME=/home/park/spark-0.7.2
source $SPARK_HOME/conf/spark-env.sh
```

In local mode, you also need to create a place for Hive to store its files, which by default is `/user/hive/warehouse`. Make sure to use the `chown` command in order to make the files accessible to your user like so:

```
mkdir -p /user/hive/warehouse && chown [your-spark-user] /user/hive/
warehouse
```

If you are using Shark with a Spark cluster, you also need to set the `MASTER` and `HADOOP_HOME` variables. If you are using Shark with an existing Hive installation, you must set `HIVE_CONF_DIR` to the directory containing the Hive XML configuration files. If you add these after the `source`... line, you can reference the variables in the Spark configuration with:

```
export HADOOP_HOME=/path/to/hadoop
export MASTER=spark://$SPARK_MASTER_IP:7077
```

Once you have Shark installed and set up, you also need to copy Shark and its custom hive to all the workers nodes; do this with:

```
pscp -v -r -h ./spark-0.7.2/conf/slaves -l sparkuser
./shark-0.7.0 ~/
pscp -v -r -h ./spark-0.7.2/conf/slaves -l sparkuser
./hive-0.9.0-bin ~/
```

If you are doing an EC2-based setup, just use the latest AMI; it should already be set up for Shark.

Running Shark

Regardless of what setup mechanism you used in the preceding section, you can launch the Shark CLI in the same way for all of them. Shark's `bin` directory provides three different variations for different levels of logging. The default of Shark, `./bin/shark`, is suitable for most cases. If you run into a problem, you may find `./bin/shark-withinfo` to be useful, and if you find problems where you need more debugging information, `./bin/shark-withdebug` is the final option. If you are connecting Shark to a Spark cluster, you should be able to see the Spark job in the web UI console under running jobs (if you don't, it is possible your Shark job is just running against a local Spark, so double check your configurations).

Loading data

Hive ships with a default dataset in `~/hive-0.9.0-bin/example/`, which you can load and use to verify if your Shark setup is working. To load the data in Shark, use the following commands:

```
shark> CREATE TABLE src(key INT, value STRING);
shark> LOAD DATA LOCAL INPATH '${env:HIVE_HOME}/examples/files/in1.txt'
INTO TABLE src;
shark> SELECT src.key, src.value FROM src WHERE src.key < 100;
```

This should output something similar to the following code, if everything was successful:

```
OK
48
Time taken: 3.02 seconds
```

Shark can also load data from S3 in the same way as Hive. To test this, you can load sample data from one of the public datasets mentioned in the HiveAWS guide, such as `s3n://data.s3ndemo.hive/kv`, for the key-value pair data. To access this, you can configure Shark with your AWS credentials in `~/hive-0.9.0-bin/conf/hive-site.xml` (you may have to create it) like so:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<configuration>

<property>
```

```
        <name>fs.s3n.awsAccessKeyId</name>
        <value>accesskeygoeshere</value>
    </property>
    <property>
        <name>fs.s3n.awsSecretAccessKey</name>
        <value>yoursecretkeygoeshere</value>
    </property>
    </configuration>
```

You can also specify your AWS credintals with `s3n://username:password@[...]` when doing your request. Assuming we have configured our AWS credentials in `hive-site.xml`, we create a table for the data like so: `create external table kv (key int, values string) location 's3n://data.s3ndemo.hive/kv';`. The HiveAWS guide explains how to load more complex data.

Shark also provides a separate Shark shell interface, which is similar to Spark's shell. In this interface, you can write a Scala code that interacts with Shark. The confusion is—the Shark context is available as `sc`, which is also used for referring `SparkContext` in the Spark shell.

Using Hive queries in a Spark program

If your data analyst (or yourself) has come up with a Hive query that you wish to use as a part of a Spark project, `sql2rdd` allows for easy integration with Scala Spark project. It is important to note that the return type is `RDD[shark.api.Row]`, so you still need to do some transformation work to make it usable by normal Spark code. The Row API provides `get[Type](rowName)` for all the supported types. So to get an integer out of a row entry, we can write `row.getInt("key")`, assuming the column with the integer is called key.

Shark was originally designed to be a standalone project and was only recently "mavenized", which allows for easy inclusion as a dependency in a similar Spark environment as covered in *Chapter 3*, *Building and Running a Spark Application*. However, Shark is not currently deployed to any of the Maven repositories, but you can deploy it to your local maven and make it available for building against it by running `sbt/sbt publish-local` in the Spark directory.

If you see errors about unsafe and an unexpected number of parameters, it is possible that sbt is using an old JDK. You can specify a specific JDK using `java_home`.

You can experiment with this using the Shark shell commands in an interactive mode. Here is a simple example of a Shark/Spark combination:

```
//Basic Shark example in Scala

package com.pandaspark.examples

import shark._
import spark.SparkContext._

object BasicSharkExample {
  def main(args: Array[String]) {
    val sc = SharkEnv.initWithSharkContext("BasicSharkExample")
    println("Starting shark requests");
    sc.sql("drop table if exists src");
    sc.sql("CREATE TABLE src(key INT, value STRING)")
    sc.sql("LOAD DATA LOCAL INPATH
           '${env:HIVE_HOME}/examples/files/in1.txt'
           INTO TABLE src")
    val rdd = sc.sql2rdd("SELECT src.key, src.value
                         FROM src WHERE src.key < 100")
    rdd.cache()
    println("Found "+rdd.count()+" num rows")
    val normalRDD = rdd.map(x => (x.getInt("src.key"),
                           x.getString("src.value")))
    println("Formatted as "+normalRDD.collect().mkString(","))
  }
}
```

You can do the same in Java:

```
//Basic Shark example in Java

package com.pandaspark.examples;

import spark.api.java.JavaRDD;
import spark.api.java.JavaPairRDD;
import spark.api.java.function.PairFunction;

import scala.Tuple2;

import shark.SharkEnv;
import shark.api.Row;
import shark.api.JavaSharkContext;
```

```
import shark.api.JavaTableRDD;

public class BasicJavaSharkExample {
  public static void main(String[] args) {
    JavaSharkContext sc = SharkEnv.
        initWithJavaSharkContext("BasicSharkExample");
    sc.sql("drop table if exists src");
    sc.sql("CREATE TABLE src(key INT, value STRING)");
    sc.sql("LOAD DATA LOCAL INPATH
           '${env:HIVE_HOME}/examples/files/in1.txt'
            INTO TABLE src");
    JavaTableRDD rdd = sc.sql2rdd("SELECT src.key,
                                src.value FROM src
                                WHERE src.key < 100");
    rdd.cache();
    System.out.println("Found "+rdd.count()+" num rows");
    JavaPairRDD<Integer, String> normalRDD = rdd.map(new
      PairFunction<Row, Integer, String>() {
      @Override
      public Tuple2<Integer, String> call(Row x) {
        return new Tuple2<Integer,String>(x.getInt("key"),
                   x.getString("value"));
      }
    });
    System.out.println("Collected: "+normalRDD.collect());
  }
}
```

Instead of depending directly on Spark, we will have our `build.sbt` pointing to:

```
libraryDependencies ++= Seq(
    "edu.berkeley.cs.amplab" % "shark_2.9.3" % "0.7.0"
)
```

You also need to include the patched version of Hive, which is distributed with Shark (while excluding an old `guava` JAR file), by adding this to the build file as well:

```
unmanagedJars in Compile <++= baseDirectory map {
  base => val hiveFile = file(System.getenv("HIVE_HOME")) / "lib"
          val baseDirectories = (base / "lib") +++ (hiveFile)
          val customJars = (baseDirectories ** "*.jar")
//Hive uses an old version of guava that doesn't have what we want
  customJars.classpath.filter(!_.toString.contains("guava"))
}
```

Running the resulting jobs is slightly different than just running normal Spark jobs. However, in the SharkEnv initialization logic, it looks at a number of environment variables to help it with the setup. As such, the easiest way to ensure you have all the correct environment variables set up is to use the provided run script and just set the `CLASSPATH` as the following code:

```
CLASSPATH=/home/spark/fastdataprocessingwithspark-sharkexamples/target/
scala-2.9.3/fastdataprocessingwithspark-sharkexamples-assembly-0.1-
SNAPSHOT.jar ./run com.pandaspark.examples.BasicSharkExample
```

Links and references

Some useful links for referencing are listed as follows:

- `http://hive.apache.org/`
- `https://github.com/amplab/shark/wiki/Shark-User-Guide`
- `https://github.com/amplab/shark/wiki`
- `https://github.com/amplab/shark/wiki/Running-Shark-Locally`
- `https://github.com/amplab/shark/wiki/Running-Shark-on-a-Cluster`
- `https://cwiki.apache.org/confluence/display/Hive/HiveAws+HivingS3nRemotely`
- `https://github.com/amplab/shark/wiki#developer-documentation`

Summary

In this chapter, you have seen how to set up Shark and how to integrate Shark into your Spark programs. In the next chapter, you will learn how to write simple unit tests.

8
Testing

Writing effective software without tests is quite challenging. Effective testing, especially in cases with slow end-to-end running times, such as distributed systems, can help improve developer effectiveness greatly. However, this chapter isn't going to try and convince you that you should be testing; if you really want to ride without a seat belt, that's fine too.

Testing in Java and Scala

For the sake of simplicity, this chapter will look at using ScalaTest and JUnit as the testing libraries. ScalaTest can be used to test both Scala and Java code and is the testing library currently used in Spark. JUnit is a popular testing framework for Java.

Refactoring your code for testability

If you have code that can be isolated from the RDD interaction or SparkContext interaction, this code can be tested using standard methodologies. While it can be quite convenient to use anonymous functions when writing Spark code, by giving them names, you can test them more easily without having to deal with the expensive overhead of setting up SparkContext. For example, in your Scala CSV parser, you could had this hard to test code:

```
val splitLines = inFile.map(line => {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
}
```

Or in Java you had:

```
JavaRDD<Integer[]> splitLines = inFile.flatMap(
new FlatMapFunction<String, Integer[]> (){
  public Iterable<Integer[]> call(String line) {
    ArrayList<Integer[]> result = new ArrayList<Integer[]>();
    try {
    CSVReader reader = new CSVReader(new StringReader(line));
    String[] parsedLine = reader.readNext();
    Integer[] intLine = new Integer[parsedLine.length];
    for (int i = 0; i < parsedLine.length; i++) {
      intLine[i] = Integer.parseInt(parsedLine[i]);
    result.add(intLine);
    }
    catch (Exception e) {
      errors.add(1);
    }
    return result;
    }
  }
);
```

Instead in Scala, you could write this in the CSV parser as a separate function:

```
def parseLine(line: String): Array[Double] = {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
}
```

While, in Java, you could add this:

```
public class JavaLoadCsvTestable {
    public static class ParseLine extends Function<String, Integer[]>
{
      public Integer[] call(String line) throws Exception {
       CSVReader reader = new CSVReader(new StringReader(line));
       String[] parsedLine = reader.readNext();
       Integer[] intLine = new Integer[parsedLine.length];
       for (int i = 0; i < parsedLine.length; i++) {
         intLine[i] = Integer.parseInt(parsedLine[i]);
       }
      return intLine;
      }
    }
```

You can then test the Java code, without having to worry about any Spark-specific setup or logic:

```
package pandaspark.examples

import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers

class TestableLoadCsvExampleSuite extends FunSuite with ShouldMatchers
{
    test("should parse a csv line with numbers") {
      TestableLoadCsvExample.parseLine("1,2") should equal
(Array[Double](1.0,2.0))
      TestableLoadCsvExample.parseLine("100,-1,1,2,2.5")
  should equal (Array[Double](100,-1,1.0,2.0,2.5))
    }
    test("should error if there is a non-number") {
      evaluating {
        TestableLoadCsvExample.parseLine("pandas")
      } should produce [NumberFormatException]
    }
}
```

Or, to test the Java code, you could write something like:

```
class JavaLoadCsvExampleSuite extends FunSuite with ShouldMatchers {

    test("should parse a csv line with numbers") {
      val parseLine = new JavaLoadCsvTestable.ParseLine();
      parseLine.call("1,2") should equal (Array[Integer](1,2))
      parseLine.call("100,-1,1,2,2") should equal
  (Array[Integer](100,-1,1,2,2))
    }
    test("should error if there is a non-integer") {
      val parseLine = new JavaLoadCsvTestable.ParseLine();
      evaluating { parseLine.call("pandas")  } should produce
  [NumberFormatException]
      evaluating {parseLine.call("100,-1,1,2.2,2") should equal
  (Array[Integer](100,-1,1,2,2)) } should produce
  [NumberFormatException]
    }
}
```

Note that the test is still written in Scala; don't worry, we will look at JUnit tests later.

Testing interactions with SparkContext

However, you may remember that you later extended our CSV parser to increment counters on invalid input so as to gracefully handle failures. To verify that behavior, you could provide mock counters and other mock objects for the Spark components you are using. You are restricted to only testing the parts of our code that don't depend on Spark. Instead, you could re-factor our code to have the core be testable without Spark as well as do a more complete test using a provided SparkContext as illustrated in the following example:

This does have the significant downside of requiring that your tests run serially as, otherwise, sbt (or another build infrastructure) may try and launch multiple SparkContext at the same time, which will cause confusing error messages. We can force tests to execute sequentially in sbt with `parallelExecution in Test := false`.

```
object MoreTestableLoadCsvExample {
  def parseLine(line: String): Array[Double] = {
    val reader = new CSVReader(new StringReader(line))
    reader.readNext().map(_.toDouble)
  }
  def handleInput(invalidLineCounter: Accumulator[Int],
  inFile: RDD[String]): RDD[Double] = {
    val numericData = inFile.flatMap(line => {
      try {
        Some(parseLine(line))
      }
      catch {
        case _ => {
        invalidLineCounter += 1
        None
        }
      }
    })
    numericData.map(row => row.sum)
  }

  def main(args: Array[String]) {
    if (args.length != 2) {
      System.err.println("Usage: TestableLoadCsvExample
  <master> <inputfile>")
      System.exit(1)
    }
    val master = args(0)
```

```
    val inputFile = args(1)
    val sc = new SparkContext(master, "Load CSV Example",
  System.getenv("SPARK_HOME"),
  Seq(System.getenv("JARS")))
    sc.addFile(inputFile)
    val inFile = sc.textFile(inputFile)
    val invalidLineCounter = sc.accumulator(0)
    val summedData = handleInput(invalidLineCounter, inFile)
    println(summedData.collect().mkString(","))
    println("Errors: "+invalidLineCounter)
    println(summedData.stats())
  }

}
```

We test this with the following code:

```
import spark._
import spark.SparkContext._
import org.scalatest.FunSuite
import org.scalatest.matchers.ShouldMatchers

class MoreTestableLoadCsvExampleSuite extends FunSuite with
ShouldMatchers {
  test("summ data on input") {
    val sc = new SparkContext("local", "Load CSV Example")
    val counter = sc.accumulator(0)
    val input = sc.parallelize(List("1,2","1,3"))
    val result = MoreTestableLoadCsvExample.handleInput(counter,
  input)
    result.collect() should equal (Array[Int](3,4))
  }
  test("should parse a csv line with numbers") {
    MoreTestableLoadCsvExample.parseLine("1,2") should equal
  (Array[Double](1.0,2.0))
    MoreTestableLoadCsvExample.parseLine("100,-1,1,2,2.5")
  should equal (Array[Double](100,-1,1.0,2.0,2.5))
  }
  test("should error if there is a non-number") {
    evaluating { MoreTestableLoadCsvExample.parseLine("pandas") }
  should produce [NumberFormatException]
  }
}
```

And in Java we use:

```
public class JavaLoadCsvMoreTestable {
    public static class ParseLineWithAcc extends
  FlatMapFunction<String, Integer[]> {
    Accumulator<Integer> acc;
    ParseLineWithAcc(Accumulator<Integer> acc) {
      this.acc = acc;
    }
    public Iterable<Integer[]> call(String line) throws Exception{
      ArrayList<Integer[]> result = new ArrayList<Integer[]>();
      try {
        CSVReader reader = new CSVReader(new StringReader(line));
        String[] parsedLine = reader.readNext();
        Integer[] intLine = new Integer[parsedLine.length];
        for (int i = 0; i < parsedLine.length; i++) {
          intLine[i] = Integer.parseInt(parsedLine[i]);
        }
        result.add(intLine);
      }
      catch (Exception e) {
        acc.add(1);
      }
      return result;
    }
  }
    public static JavaDoubleRDD processData(
  Accumulator<Integer> acc, JavaRDD<String> input) {
      JavaRDD<Integer[]> splitLines = input.flatMap(
  new ParseLineWithAcc(acc));
      JavaDoubleRDD summedData = splitLines.map(
  new DoubleFunction<Integer[]>() {
        public Double call(Integer[] in) {
          Double ret = 0.;
          for (int i = 0; i < in.length; i++) {
            ret += in[i];
          }
        return ret;
        }
      }
      );
    return summedData;
}
```

You can test this in Scala with:

```
class JavaLoadCsvMoreTestableSuite extends FunSuite
with ShouldMatchers {
  test("sum data on input") {
    val sc = new JavaSparkContext("local", "Load Java CSV test")
    val counter: Accumulator[Integer] = sc.intAccumulator(0)
    val input: JavaRDD[String] = sc.parallelize
  (List("1,2","1,3","murh"))
    val javaLoadCsvMoreTestable = new JavaLoadCsvMoreTestable();
    val resultRDD = JavaLoadCsvMoreTestable.
  processData(counter,input)
    resultRDD.cache();
    val resultCount = resultRDD.count()
    val result = resultRDD.collect().toArray()
    resultCount should equal (2)
    result should equal (Array[Double](3.0, 4.0))
    counter.value should equal (1)
    sc.stop()
  }
}
```

Note that we add an invalid input for the counter.

In Java, using JUnit4 you can add the following code for testing:

```
package pandaspark.examples;

import spark.*;
import spark.api.java.JavaSparkContext;
import spark.api.java.JavaRDD;
import spark.api.java.JavaDoubleRDD;
import org.scalatest.FunSuite;
import org.scalatest.matchers.ShouldMatchers;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.Ignore;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
```

```
@RunWith(JUnit4.class)
public class JavaLoadCsvMoreTestableSuiteJunit {
    @Test
    public void testSumDataOnInput() {
      JavaSparkContext sc = new JavaSparkContext("local",
  "Load Java CSV test");
      Accumulator<Integer> counter = sc.intAccumulator(0);
      String[] inputArray = {"1,2","1,3","murh"};
      JavaRDD<String> input = sc.parallelize
  (Arrays.asList(inputArray));
      JavaDoubleRDD resultRDD = JavaLoadCsvMoreTestable.
  processData(counter, input);
      long resultCount = resultRDD.count();
      assertEquals(resultCount, 2);
      int errors = counter.value();
      assertEquals(errors, 1);
      sc.stop();
    }
}
```

Testing in Python

Python testing of Spark is very similar in concept, but the testing libraries are a bit different. PySpark uses both `doctest` and `unittest` to test itself. The `doctest` library makes it easy to create tests based on the expected output of code run in the Python interpreter. We can run the tests by running `pyspark -m doctest [pathtocode]`. By taking the `wordcount.py` example from Spark and factoring out `countWords`, you can test the word count functionality using `doctest`:

```
"""
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.parallelize(["pandas are awesome", "and ninjas are also
awesome"])
>>> countWords(b)
[('also', 1), ('and', 1), ('are', 2), ('awesome', 2), ('ninjas', 1),
('pandas', 1)]
"""

import sys
from operator import add

from pyspark import SparkContext
```

```
def countWords(lines):
    counts = lines.flatMap(lambda x: x.split(' ')) \
                  .map(lambda x: (x, 1)) \
                  .reduceByKey(add)
    return sorted(counts.collect())


if __name__ == "__main__":
    if len(sys.argv) < 3:
      print >> sys.stderr, "Usage: PythonWordCount
  <master> <file>"
      exit(-1)
    sc = SparkContext(sys.argv[1], "PythonWordCount")
    lines = sc.textFile(sys.argv[2], 1)
    output = countWords(lines)
    for (word, count) in output:
      print "%s : %i" % (word, count)
```

We can also test something similar to our Java and Scala programs like so:

```
"""
>>> from pyspark.context import SparkContext
>>> sc = SparkContext('local', 'test')
>>> b = sc.parallelize(["1,2","1,3"])
>>> handleInput(b)
[3, 4]
"""

import sys
from operator import add

from pyspark import SparkContext
def handleInput(lines):
    data = lines.map(lambda x: sum(map(int, x.split(','))))
    return sorted(data.collect())


if __name__ == "__main__":
    if len(sys.argv) < 3:
        print >> sys.stderr, "Usage: PythonLoadCsv
  <master> <file>"
        exit(-1)
    sc = SparkContext(sys.argv[1], "PythonLoadCsv")
    lines = sc.textFile(sys.argv[2], 1)
    output = handleInput(lines)
    for sum in output:
      print sum
```

Links and references

Here are some useful links for reference:

- `http://blog.quantifind.com/posts/spark-unit-test/`
- `http://www.scalatest.org/`
- `http://junit.org/`
- `http://docs.python.org/2/library/unittest.html`
- `http://docs.python.org/2/library/doctest.html`

Summary

This chapter has looked at how to structure code so that it is testable and at the testing framework that is used within Spark. Effective testing can save large amounts of debugging time, which can be especially painful in large distributed systems. In the next chapter, we will look at some tips and tricks, such as tuning and securing Spark.

9
Tips and Tricks

Now that you have the tools to build and test Spark jobs as well as set up a Spark cluster to run them on, it's time to figure out how to make the most of your time as a Spark developer.

Where to find logs?

Spark and Shark have very useful logs for figuring out what's going on when things are not behaving as expected. When working with a program that uses `sql2rdd` or any other Shark-related tool, a good place to start debugging is by looking at what HiveQL queries are being run. You should find this in the console logs where you execute the Spark program: look for a line such as **Hive history file=/tmp/spark/hive_job_log_spark_201306090132_919529074.txt**. Spark also keeps a per machine log on each machine, by default, in the logs subdirectory of the Spark directory. Spark's web UI provides a convenient place to see the `stdout` and `stderr` files of each job, running and completing separate output per worker.

Concurrency limitations

Spark's concurrency for operations is limited by the number of partitions. Conversely, having too many partitions can cause an excess overhead with too many tasks being launched. If you have too many partitions, you can shrink it down using the `coalesce(count)` method; `coalesce` will only decrease the number of partitions. When creating a new RDD, you can specify the number of splits to be used. Also, the grouping/joining mechanism on the RDDs of pairs can take the number of partitions or, alternatively, a partitioner. The default number of partitions for new RDDs is controlled by `spark.default.parallelism`, which also controls the number of tasks used by `groupByKey` and other shuffle operations.

Memory usage and garbage collection

To measure the impact of garbage collection, you can ask the JVM to print details about the garbage collection. You can do this by adding `-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` to your `SPARK_JAVA_OPTS` environment variable in `conf/spark-env.sh`. The details will then be printed to the standard output when you run your job, which will be available as described in the *Where to find logs?* section of this chapter.

If you find that your Spark cluster is using too much time on garbage collection, you can reduce the amount of space used for RDD caching by changing `spark.storage.memoryFraction`, which is set to `0.66` by default. If you are planning to run Spark for a long time on a cluster, you may wish to enable `spark.cleaner.ttl`. By default, Spark does not clean up any metadata; set `spark.cleaner.ttl` to a nonzero value in seconds to clean up metadata after that length of time.

You can also control the RDD storage level if you find that you are using too much memory. If your RDDs don't fit in the memory and you still wish to cache them, you can try using a different storage level such as:

- `MEMORY_ONLY`: This stores the entire RDD in the memory if it can and is the default storage level
- `MEMORY_AND_DISK`: This stores each partition in the memory if it can, or if it doesn't, it stores it on disk
- `DISK_ONLY`: This stores each partition on the disk regardless of whether it can fit in the memory

These options are set when you call the persist function on your RDD. By default, the RDDs are stored in a deserialized form, which requires less parsing. We can save space by adding `_SER` to the storage level; in this case, Spark will serialize the data to be stored, which normally saves some space.

Serialization

Spark supports different serialization mechanisms; the choice is a trade-off between speed, space efficiency, and full support of all Java objects. If you are using a serializer to cache your RDDs, you should strongly consider a fast serializer. The default serializer uses Java's default serialization. The KyroSerializer is much faster and generally uses about one-tenth of the memory as the default serializer. You can switch the serializer by changing `spark.serializer` to `spark.KryoSerializer`. If you want to use the KyroSerializer, you need to make sure that the classes are serializable by the KyroSerializer.

Spark provides a trait KryoRegistrator, which you can extend to register your classes with Kyro as follows:

```
class MyReigstrator extends spark.KyroRegistrator {
   override def registerClasses(kyro: Kyro) {
    kyro.register(classOf[MyClass])
   }
}
```

Visit `https://code.google.com/p/kryo/#Quickstart` to figure out how to write custom serializers for your classes if you need something customized. You can substantially decrease the amount of space used for your objects by customizing your serializers. For example, rather than writing out the full class name, you can give them an integer ID by calling `kyro.register(classOf[MyClass],100)`.

IDE integration

As an Emacs user, the author finds that having an **ENhanced Scala Interaction Mode** (**ensime**) setup helps with development. You can install the latest ensime from `https://github.com/aemoncannon/ensime/downloads` (make sure to choose the one that matches your Scala version).

```
wget https://github.com/downloads/aemoncannon/ensime/ensime_2.9.2-
0.9.8.1.tar.gz
tar -xvf ensime_2.9.2-0.9.8.1.tar.gz
```

In your `.emacs` file, add:

```
;; Load the ensime lisp code...
(add-to-list 'load-path "ENSIME_ROOT/elisp/")
(require 'ensime);;
This step causes the ensime-mode to be started whenever
;; scala-mode is started for a buffer. You may have to customize this
step
;; if you're not using the standard scala mode.
(add-hook 'scala-mode-hook 'ensime-scala-mode-hook)
```

You can then add the `ensime sbt` plugin to your project (in `project/plugins.sbt`):

```
addSbtPlugin("org.ensime" % "ensime-sbt-cmd" % "0.1.0")
```

You can then run the plugin:

```
sbt
> ensime generate
```

If you are using git, you will probably want to add `.ensime` to the `.gitignore` file if it isn't already present.

If you are using IntelliJ, a similar plugin exists called `sbt-idea` that can be used to generate IntelliJ IDEA files. You can add the `IntelliJ sbt` plugin to your project (in `project/plugins.sbt`):

```
addSbtPlugin("com.github.mpeltonen" % "sbt-idea" % "1.5.1")
```

You can then run the plugin:

```
sbt
> gen-idea
```

This will generate the IDEA project file that can be loaded into IntelliJ.

Eclipse users can also use `sbt` to generate Eclipse project files with the `sbteclipse` plugin. You can add the Eclipse sbt plugin to your project (in `project/plugins.sbt`):

```
addSbtPlugin("com.typesafe.sbteclipse" % "sbteclipse-plugin" %
"2.3.0")
```

You can then run the plugin:

```
sbt
> eclipse
```

This will generate the Eclipse project files, and you can then import them into your Eclipse project using the **Import** wizard in Eclipse. Eclipse users might also find the `spark-plug` project useful; it can be used to launch clusters from within Eclipse.

Using Spark with other languages

If you find yourself wanting to work with your RDD in another language, there are a few options. With Java/Scala, you can try using the JNI, and with Python, you can use the FFI. Sometimes, however, you will want to work with a language that isn't C language or with an already compiled program. In that case, the easiest thing to do is use the pipe interface that is available in all of the three APIs. The Stream API works by taking the RDD, serializing it to strings, and piping it to the specified program. If your data happens to be plain strings, this is very convenient; but if not, you will need to serialize your data in such a way it can be understood on either side. JSON or protocol buffers can be good options depending on how structured your data is.

A quick note on security

Another important consideration in your Spark setup is security. If you are using Spark on EC2 with the default scripts, you will notice that access to your Spark cluster is restricted. This is a good idea even if you aren't running Spark inside EC2 since your Spark cluster will most likely have access to data you would rather not share with the world. (And even if it doesn't, you probably don't want to allow arbitrary code execution by strangers.) If your Spark cluster is already on a private network, that's great; otherwise, you should talk to your system's administrator about setting up some IPTables rules to restrict access.

Mailing lists

Probably the most useful tip to finish with is that the Spark-users' mailing list is an excellent source of up-to-date information about other people's experiences with Spark. You can subscribe to `https://groups.google.com/forum/?fromgroups#!forum/spark-users` (soon to be `http://mail-archives.apache.org/mod_mbox/incubator-spark-user/`) as well as search the archives to see if other people have run into similar problems as you have.

Links and references

Some useful links for referencing are listed as follows:

- `http://blog.quantifind.com/posts/logging-post/`
- `http://jawher.net/2011/01/17/scala-development-environment-emacs-sbt-ensime/`
- `https://www.assembla.com/spaces/liftweb/wiki/Emacs-ENSIME`
- `http://syndeticlogic.net/?p=311`
- `http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf`
- `https://github.com/shivaram/spark-ec2/blob/master/ganglia/init.sh`
- `http://spark-project.org/docs/0.7.2/tuning.html`
- `https://github.com/mesos/spark/blob/master/docs/configuration.md`
- `http://kryo.googlecode.com/svn/api/v2/index.html`
- `https://code.google.com/p/kryo/`

- `http://scala-ide.org/download/current.html`
- `http://syndeticlogic.net/?p=311`
- `http://mail-archives.apache.org/mod_mbox/incubator-spark-user/`
- `https://groups.google.com/forum/?fromgroups#!forum/spark-users`

Summary

That wraps up some common things, which you can use to help improve your Spark development experience. I wish you the best of luck with your Spark projects. Now go solve some fun problems!

Index

A

addFile(path) method 41
addJar(path) method 41
aggregate function 64
AMI (Amazon Machine Images) 12

B

bin/slaves.sh <command> command 19
bin/start-all.sh command 19
bin/start-master.sh command 19
bin/start-slave.sh command 19
bin/start-slaves.sh command 19
bin/stop-all.sh command 19
bin/stop-master.sh command 20
bin/stop-slaves.sh command 20

C

cache function 64, 68
cartesian function 73
Chef (opscode)
  used, for Spark deploying 14, 15
chown command 78
clearFiles() method 41
clearJars() method 41
coalesce function 68
code
  testing 85-87
cogroup function 61, 70, 76
collectAsMap function 62, 70, 75
collect function 64, 68
collect() function 46
combineByKey function 70, 76
concurrency
  limitations 95
countByKey function 62, 70, 75
countByValue function 64, 68, 74
count function 64, 68

D

data
  loading 79, 80
distinct function 64, 68, 73
DoubleFlatMapFunction function 67
DoubleFunction<T> function 67
DoubleRDD functions
  mean 64
  sampleStdev 64
  Stats 64
  Stdev 64
  Sum 64
  variance 64

E

EC2
  Spark, running on 8
Elastic MapReduce
  Spark, deploying on 13
ENhanced Scala Interaction Mode (ensime) 97

F

filter function 68, 73
filterWith function 65
first function 65, 68
flatMap function 65, 68, 73
FlatMapFunction<T, R> function 67
flatMap method 56
flatMapValues function 63, 71

foldByKey function 60
fold function 65, 68, 74
foreach function 65, 68, 74
Function2<T1, T2, R> function 67
Function<T,R> function 67

G

garbage collection
impact, measuring 96
general RDD functions
aggregate 64
cache 64
collect 64
count 64
countByValue 64
distinct 64
filter 65
filterWith 65
first 65
flatMap 65
fold 65
foreach 65
groupBy 65
keyBy 65
map 65
mapPartitions 65
mapPartitionsWithIndex 65
mapWith 66
persist 66
pipe 66
sample 66
takeSample 66
toDebugString 66
union 66
unpersist 66
zip 66
groupBy function 65, 69, 74
groupByKey function 61, 76

H

Hadoop Distributed File System. *See* **HDFS**
HBase database
use 47-49
HDFS 5
Hive 77

I

IDE integration 97, 98
installation
Shark 78
IntelliJ sbt plugin 98

J

Java
SparkContext, creating in 40
testing 85
JavaPairRDD functions
cogroup 70
collectAsMap 70
combineByKey 70
countByKey 70
flatMapValues 71
join 71
keys 71
lookup 71
reduceByKey 71
sortByKey 71
values 71
JavaPairRDD functions combination methods
subtract 69
union 69
zip 70
Java RDD functions
cache 68
coalesce 68
collect 68
count 68
countByValue 68
distinct 68
filter 68
first 68
flatMap 68
fold 68
foreach 68
groupBy 69
map 69
mapPartitions 69
reduce 69
sample 69
Spark Java function classes 67

join function 61, 71, 75
joining functions, for PairRDD functions
cogroup function 61
join function 61
subtractKey function 61

K

keyBy function 65
keys function 71

L

leftOuterJoin function 75
logs
finding 95
lookup function 62, 71

M

map function 65, 69
mapPartitions function 65, 69, 73
mapPartitionsWithIndex function 65
mapValues function 62
mapWith function 66
Maven
used, for Spark job building 35-37
mean function 64
Mesos
used, for Spark deploying 15
MESOS_NATIVE_LIBRARY variable 17

N

newAPIHadoopRDD method 49

P

PairFlatMapFunction<T, K, V> function 67
PairFunction<T, K, V> function 67
PairRDD functions, for RDD manipulation in Java
collectAsMap 62
countByKey 62
flatMapValues 63
lookup 62
mapValues 62
partitionBy 63
PairRDD functions, for RDD manipulation in Python
cogroup 76
collectAsMap 75
combineByKey 76
countByKey 75
groupByKey 76
join 75
leftOuterJoin 75
reduceByKey 75
rightOuterJoin 75
parallelize() function 46
partitionBy function 63, 74
persist function 66
pipe function 66, 74
PPA (Personal Package Archive) 32
PRNG (pseudorandom number generator) 60
Python
RDD, manipulating in 71, 72
SparkContext, creating in 41
testing 92, 93

R

RDD manipulation, in Python
about 71, 72
PairRDD functions 75
standard RDD functions 73
RDDs
about 43
data, loading into 44-49
manipulating, in Java 51
manipulating in Python 71
manipulating, in Scala 51
saving, ways 49
reduceByKey function 61, 71, 75
reduce function 69, 74
reference links
about 21, 76
Hive queries, using in 83
mailing lists 99, 100
saving 49, 50
SparkContext, creating in 42
Resilient Distributed Datasets. See RDDs
rightOuterJoin function 75

S

S3
  data, loading from 27, 28
  path 27
sample function 66, 69
sampleStdev function 64
sbt
  used,for Spark project building 31-34
Scala
  about 40
  SparkContext, creating in 40
  testing 85
SCALA_HOME variable 17
Scala RDD functions
  foldByKey 60
  groupByKey 61
  reduceByKey 61
Scala REPL (Read-Evaluate-Print Loop) 26
serialization mechanisms 96, 97
shared Java APIs 41
shared Scala APIs 41
Shark
  about 77
  installing 78
  running 79
simple text file
  loading 23, 24
sortByKey function 71
Spark
  about 7
  deploying, Chef (opscode) 14, 15
  deploying, on Elastic MapReduce 13
  deploying, on Mesos 15, 16
  deploying, on YARN 16
  deploying, over SSH 17-19
  mailing lists 99
  running, on EC2 8
  running, on EC2 with scripts 8-12
  running, on single machine 7
  security 99
  using, with other languages 98
SparkContext
  about 39
  application name 40
  creating, in Java 40
  creating, in Python 41
  creating, in Scala 40
  interactions, testing 88-92
  jars 40
  master 40
  sparkHome 40
SparkContext class 49
Spark Java function classes
  DoubleFlatMapFunction<T> 67
  DoubleFunction<T> 67
  FlatMapFunction<T, R> 67
  Function2<T1, T2, R> 67
  Function<T,R> 67
  PairFlatMapFunction<T, K, V> 67
  PairFunction<T, K, V> 67
Spark job
  building, with Maven 35-37
  building, with other options 37
SPARK_MASTER_IP variable 17
SPARK_MASTER_PORT variable 18
SPARK_MASTER_WEBUI_PORT variable 18
Spark program
  Hive queries, using in 80-82
Spark project
  building, with sbt 31-34
Spark RDDs. *See* **RDDs**
Spark shell
  about 23
  used, for logistic regression running 25, 26
SPARK_WEBUI_PORT variable 18
SPARK_WORKER_CORES variable 18
SPARK_WORKER_DIR variable 18
SPARK_WORKER_MEMORY variable 18
SPARK_WORKER_PORT variable 18
standalone mode 17
Stats function 64
Stdev function 64
stop() method 41
storage level
  DISK_ONLY 96
  MEMORY_AND_DISK 96
  MEMORY_ONLY 96
subtract function 69
subtractKey function 61
Sum function 64

T

take function 74
takeSample function 66
testing
in Java 85
in Python 92, 93
in Scala 85
reference links 94
toDebugString function 66

U

union function 66, 69, 73
unpersist function 66

V

values function 71
variance function 64

Y

YARN
used, for Spark deploying 16

Z

zip function 66, 70

Thank you for buying

Fast Data Processing with Spark

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

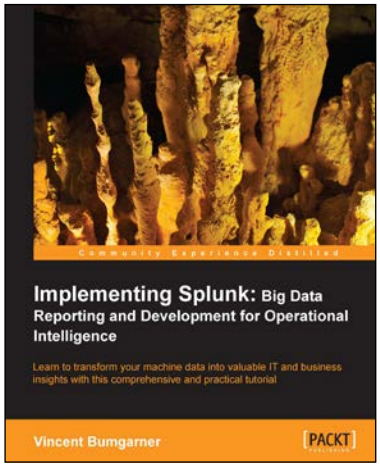
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



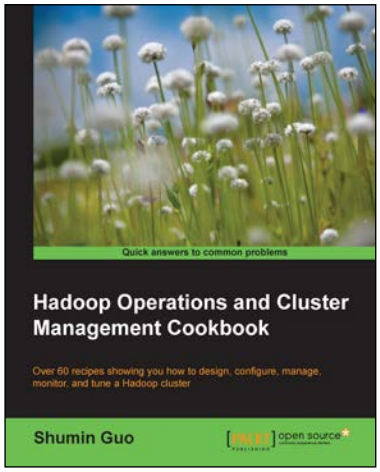
Implementing Splunk: Big Data Reporting and Development for Operational Intelligence

ISBN: 978-1-849693-28-8 Paperback: 448 pages

Learn to transform your machine data into valuable IT and business insights with this comprehensive and practical tutorial

1. Learn to search, dashboard, configure, and deploy Splunk on one machine or thousands
2. Start working with Splunk fast, with a tested set of practical examples and useful advice
3. Step-by-step instructions and examples with a comprehensive coverage for Splunk veterans and newbies alike

---



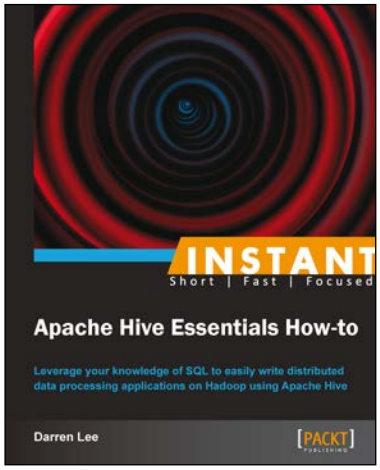
Hadoop Operations and Cluster Management Cookbook

ISBN: 978-1-782165-16-3 Paperback: 368 pages

Over 60 recipes showing you how to design, configure, manage, monitor, and tune a Hadoop cluster

1. Hands-on recipes to configure a Hadoop cluster from bare metal hardware nodes
2. Practical and in depth explanation of cluster management commands
3. Easy-to-understand recipes for securing and monitoring a Hadoop cluster, and design considerations

[PACKT] open source
community experience distilled
PUBLISHING



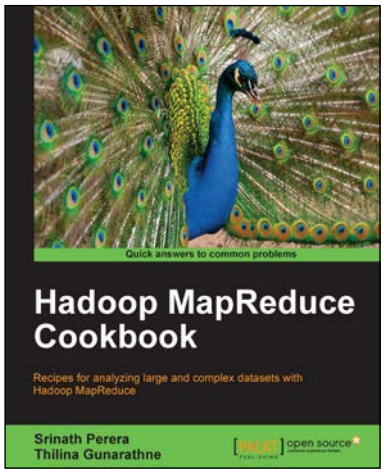
Instant Apache Hive Essentials How-to

ISBN: 978-1-782169-47-5 Paperback: 76 pages

Leverage your knowledge of SQL to easily write distributed data processing applications on Hadoop using Apache Hive

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Learn to use SQL to write Hadoop jobs
3. Understand how the Hive query processor works to optimize common queries

---



Hadoop MapReduce Cookbook

ISBN: 978-1-849517-28-7 Paperback: 300 pages

Recipes for analyzing large and complex datasets with Hadoop MapReduce

1. Learn to process large and complex data sets, starting simply, then diving in deep
2. Solve complex big data problems such as classifications, finding relationships, online marketing and recommendations
3. More than 50 Hadoop MapReduce recipes, presented in a simple and straightforward manner, with step-by-step instructions and real world examples