O'REILLY®

Learning Spark

LIGHTNING-FAST DATA ANALYSIS

Holden Karau,
Andy Kowinski & Matei Zaharia

Learning Spark

Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Learning Spark

by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Ann Spencer and Marie Beaugureau
Production Editor: Kara Ebrahim
Copyeditor: FIX ME
Proofreader: FIX ME!

Indexer: FIX ME
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

February 2015: First Edition

Revision History for the First Edition:

2014-06-23: First release

2014-07-28: Second release

2014-09-17: Third release

2014-09-23: Fourth release

2014-11-12: Fifth release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449358624* for release details.

ISBN: 978-1-449-35862-4

[?]

Table of Contents

Preface

As parallel data analysis has become increasingly common, practitioners in many fields have sought easier tools for this task. Apache Spark has quickly emerged as one of the most popular tools for this purpose, extending and generalizing MapReduce. Spark offers three main benefits. First, it is easy to use — you can develop applications on your laptop, using a high-level API that lets you focus on the content of your computation. Second, Spark is fast, enabling interactive use and complex algorithms. And third, Spark is a *general* engine, allowing you to combine multiple types of computations (e.g., SQL queries, text processing and machine learning) that might previously have required learning different engines. These features make Spark an excellent starting point to learn about big data in general.

This introductory book is meant to get you up and running with Spark quickly. You'll learn how to download and run Spark on your laptop and use it interactively to learn the API. Once there, we'll cover the details of available operations and distributed execution. Finally, you'll get a tour of the higher-level libraries built into Spark, including libraries for machine learning, stream processing, graph analytics and SQL. We hope that this book gives you the tools to quickly tackle data analysis problems, whether you do so on one machine or hundreds.

Audience

This book targets Data Scientists and Engineers. We chose these two groups because they have the most to gain from using Spark to expand the scope of problems they can solve. Spark's rich collection of data focused libraries (like MLlib) make it easy for data scientists to go beyond problems that fit on single machine while making use of their statistical background. Engineers, meanwhile, will learn how to write general-purpose distributed programs in Spark and operate production applications. Engineers and data scientists will both learn different details from this book, but will both be able to apply Spark to solve large distributed problems in their respective fields.

Data scientists focus on answering questions or building models from data. They often have a statistical or math background and some familiarity with tools like Python, R and SQL. We have made sure to include Python, and wherever possible SQL, examples for all our material, as well as an overview of the machine learning and advanced analytics libraries in Spark. If you are a data scientist, we hope that after reading this book you will be able to use the same mathematical approaches to solving problems, except much faster and on a much larger scale.

The second group this book targets is software engineers who have some experience with Java, Python or another programming language. If you are an engineer, we hope that this book will show you how to set up a Spark cluster, use the Spark shell, and write Spark applications to solve parallel processing problems. If you are familiar with Hadoop, you have a bit of a head start on figuring out how to interact with HDFS and how to manage a cluster, but either way, we will cover basic distributed execution concepts.

Regardless of whether you are a data analyst or engineer, to get the most of this book you should have some familiarity with one of Python, Java, Scala, or a similar language. We assume that you already have a solution for storing your data and we cover how to load and save data from many common ones, but not how to set them up. If you don't have experience with one of those languages, don't worry, there are excellent resources available to learn these. We call out some of the books available in Supporting Books.

How This Book is Organized

The chapters of this book are laid out in such a way that you should be able to go through the material front to back. At the start of each chapter, we will mention which sections of the chapter we think are most relevant to data scientists and which sections we think are most relevant for engineers. That said, we hope that all the material is accessible to readers of either background.

The first two chapters will get you started with getting a basic Spark installation on your laptop and give you an idea of what you can accomplish with Apache Spark. Once we've got the motivation and setup out of the way, we will dive into the Spark Shell, a very useful tool for development and prototyping. Subsequent chapters then cover the Spark programming interface in detail, how applications execute on a cluster, and higher-level libraries available on Spark such as Spark SQL and MLlib.

Supporting Books

If you are a data scientist and don't have much experience with Python, the Learning Python and Head First Python books are both excellent introductions. If you have some Python experience and want some more, Dive into Python is a great book to get a deeper understanding of Python.

If you are an engineer and after reading this book you would like to expand your data analysis skills, Machine Learning for Hackers and Doing Data Science are excellent books from O'Reilly.

This book is intended to be accessible to beginners. We do intend to release a deep dive follow-up for those looking to gain a more thorough understanding of Spark's internals.

Code Examples

All of the code examples found in this book are on GitHub. You can examine them and check them out from *https://github.com/databricks/learning-spark*. Code examples are provided in Java, Scala, and Python.

Our Java examples are written to work with Java version 6 and higher. Java 8 introduces a new syntax called "lambdas" that makes writing inline functions much easier, which can simplify Spark code. We have chosen not to take advantage of this syntax in most of our examples, as most organizations are not yet using Java 8. If you would like to try Java 8 syntax, you can see the Databricks blog post on this topic. Some of the examples will also be ported to Java 8 and posted to the books GitHub.

Early Release Status and Feedback

This is an **early release** copy of Learning Spark, and as such we are still working on the text, adding code examples, and writing some of the later chapters. Although we hope that the book is useful in its current form, we would greatly appreciate your feedback so we can improve it and make the best possible finished product. The authors and editors can be reached at *book-feedback@databricks.com*.

The authors would like to thank the reviewers who offered feedback so far: Juliet Hougland, Andrew Gal, Michael Gregson, Stephan Jou, Josh Mahonin, Mike Patterson, and Deborah Siegel.

We would also like to thank the subject matter experts who took time to edit and write parts of their own chapters. Tathagata Das worked with us on a very tight schedule to get the Spark Streaming chapter finished. Tathagata went above and beyond with clarifying examples, answering many questions, and improving the flow of the text in addition to his technical contributions.

CHAPTER 1

Introduction to Data Analysis with Spark

This chapter provides a high level overview of what Apache Spark is. If you are already familiar with Apache Spark and its components, feel free to jump ahead to Chapter 2.

What is Apache Spark?

Apache Spark is a cluster computing platform designed to be *fast* and *general-purpose*.

On the speed side, Spark extends the popular MapReduce model to efficiently support more types of computations, including interactive queries and stream processing. Speed is important in processing large datasets as it means the difference between exploring data interactively and waiting minutes between queries, or waiting hours to run your program versus minutes. One of the main features Spark offers for speed is the ability to run computations in memory, but the system is also faster than MapReduce for complex applications running on disk.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to *combine* different processing types, which is often necessary in production data analysis pipelines. In addition, it reduces the management burden of maintaining separate tools.

Spark is designed to be highly accessible, offering simple APIs in Python, Java, Scala and SQL, and rich built-in libraries. It also integrates closely with other big data tools. In particular, Spark can run in Hadoop clusters and access any Hadoop data source, including Cassandra.

A Unified Stack

The Spark project contains multiple closely-integrated components. At its core, Spark is a "computational engine" that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across many worker machines, or a *computing cluster*. Because the core engine of Spark is both fast and general-purpose, it powers multiple higher-level components specialized for various workloads, such as SQL or machine learning. These components are designed to interoperate closely, letting you combine them like libraries in a software project.

A philosophy of tight integration has several benefits. First, all libraries and higher level components in the stack benefit from improvements at the lower layers. For example, when Spark's core engine adds an optimization, SQL and machine learning libraries automatically speed up as well. Second, the costs associated with running the stack are minimized, because instead of running 5-10 independent software systems, an organization only needs to run one. These costs include deployment, maintenance, testing, support, and more. This also means that each time a new component is added to the Spark stack, every organization that uses Spark will immediately be able to try this new component. This changes the cost of trying out a new type of data analysis from downloading, deploying, and learning a new software project to upgrading Spark.

Finally, one of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models. For example, in Spark you can write one application that uses machine learning to classify data in real time as it is ingested from streaming sources. Simultaneously analysts can query the resulting data, also in real-time, via SQL, e.g. to join the data with unstructured log files. In addition, more sophisticated data engineers & data scientists can access the same data via the Python shell for ad-hoc analysis. Others might access the data in standalone batch applications. All the while, the IT team only has to maintain one software stack.

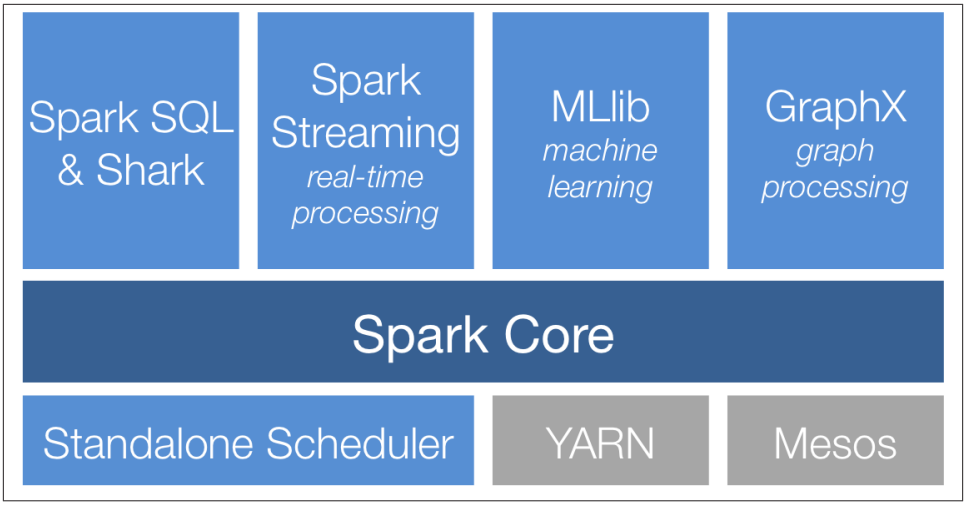

Figure 1-1. The Spark Stack

Here we will briefly introduce each of the components shown in Figure 1-1.

Spark Core

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines Resilient Distributed Datasets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections.

Spark SQL

Spark SQL provides support for interacting with Spark via SQL as well as the Apache Hive variant of SQL, called the Hive Query Language (HiveQL). Spark SQL represents database tables as Spark RDDs and translates SQL queries into Spark operations. Beyond providing the SQL interface to Spark, Spark SQL allows developers to intermix SQL queries with the programmatic data manipulations supported by RDDs in Python, Java and Scala, all within a single application. This tight integration with the rich and sophisticated computing environment provided by the rest of the Spark stack makes Spark SQL unlike any other open source data warehouse tool. Spark SQL was added to Spark in version 1.0.

Shark is a project out of UC Berkeley that predates Spark SQL and is being ported to work on top of Spark SQL. Shark provides additional functionality so that Spark can

act as drop-in replacement for Apache Hive. This includes a HiveQL shell, as well as a JDBC server that makes it easy to connect external graphing and data exploration tools.

Spark Streaming

Spark Streaming is a Spark component that enables processing live streams of data. Examples of data streams include log files generated by production web servers, or queues of messages containing status updates posted by users of a web service. Spark Streaming provides an API for manipulating data streams that closely matches the Spark Core's RDD API, making it easy for programmers to learn the project and move between applications that manipulate data stored in memory, on disk, or arriving in real-time. Underneath its API, Spark Streaming was designed to provide the same degree of fault tolerance, throughput, and scalability that the Spark Core provides.

MLlib

Spark comes with a library containing common machine learning (ML) functionality called MLlib. MLlib provides multiple types of machine learning algorithms, including binary classification, regression, clustering and collaborative filtering, as well as supporting functionality such as model evaluation and data import. It also provides some lower level ML primitives including a generic gradient descent optimization algorithm. All of these methods are designed to scale out across a cluster.

GraphX

GraphX is a library added in Spark 0.9 that provides an API for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. GraphX also provides set of operators for manipulating graphs (e.g., `subgraph` and `mapVertices`) and a library of common graph algorithms (e.g., PageRank and triangle counting).

Cluster Managers

Cluster Managers are a bit different as the previous components are things that are built on Spark, but Spark can run on different cluster managers. Under the hood, Spark is designed to efficiently scale up from one to many thousands of compute nodes. To achieve this while maximizing flexibility, Spark can run over a variety of *cluster managers*, including Hadoop YARN, Apache Mesos, and a simple cluster manager included in Spark itself called the Standalone Scheduler. If you are just installing Spark on an empty set of machines, the Standalone Scheduler provides an easy way to get started; while if you already have a Hadoop YARN or Mesos cluster, Spark's support for these

allows your applications to also run on them. The Chapter 7 explores the different options and how to choose the correct cluster manager.

Who Uses Spark, and For What?

Because Spark is a general purpose framework for cluster computing, it is used for a diverse range of applications. In the Preface we outlined two personas that this book targets as readers: Data Scientists and Engineers. Let's take a closer look at each of these personas and how they use Spark. Unsurprisingly, the typical use cases differ across the two personas, but we can roughly classify them into two categories, *data science* and *data applications*.

Of course, these are imprecise personas and usage patterns, and many folks have skills from both, sometimes playing the role of the investigating Data Scientist, and then "changing hats" and writing a hardened data processing system. Nonetheless, it can be illuminating to consider the two personas and their respective use cases separately.

Data Science Tasks

Data Science is the name of a discipline that has been emerging over the past few years centered around analyzing data. While there is no standard definition, for our purposes a *Data Scientist* is somebody whose main task is to analyze and model data. Data scientists may have experience using SQL, statistics, predictive modeling (machine learning), and some programming, usually in Python, Matlab or R. Data scientists also have experience with techniques necessary to transform data into formats that can be analyzed for insights (sometimes referred to as *data wrangling*).

Data Scientists use their skills to analyze data with the goal of answering a question or discovering insights. Oftentimes, their workflow involves ad-hoc analysis, and so they use interactive shells (vs. building complex applications) that let them see results of queries and snippets of code in the least amount of time. Spark's speed and simple APIs shine for this purpose, and its built-in libraries mean that many algorithms are available out of the box.

Spark supports the different tasks of data science with a number of components. The PySpark shell makes it easy to do interactive data analysis using Python. Spark SQL also has a separate SQL shell which can be used to do data exploration using SQL, or Spark SQL can be used as part of a regular Spark program or in the PySpark shell. Machine learning and data analysis is supported through the MLLib libraries. In addition support exists for calling out to existing programs in Matlab or R. Spark enables Data Scientists to tackle problems with larger data sizes than they could before with tools like R or Pandas.

Sometimes, after the initial exploration phase, the work of a Data Scientist will be "productionized", or extended, hardened (i.e. made fault tolerant), and tuned to become a

production data processing application, which itself is a component of a business application. For example, the initial investigation of a Data Scientist might lead to the creation of a production recommender system that is integrated into a web application and used to generate customized product suggestions to users. Often it is a different person or team that leads the process of productizing the work of the Data Scientists, and that person is often an Engineer.

Data Processing Applications

The other main use case of Spark can be described in the context of the *Engineer* persona. For our purposes here, we think of Engineers as large class of software developers who use Spark to build production data processing applications. These developers usually have an understanding of the principles of software engineering, such as encapsulation, interface design, and Object Oriented Programming. They frequently have a degree in Computer Science. They use their engineering skills to design and build software systems that implement a business use case.

For Engineers, Spark provides a simple way to parallelize these applications across clusters, and hides the complexity of distributed systems programming, network communication and fault tolerance. The system gives enough control to monitor, inspect and tune applications while allowing common tasks to be implemented quickly. The modular nature of the API (based on passing distributed collections of objects) makes it easy to factor work into reusable libraries and test it locally.

Spark's users choose to use it for their data processing applications because it provides a wide variety of functionality, is easy to learn and use, and is mature and reliable.

A Brief History of Spark

Spark is an open source project that has been built and is maintained by a thriving and diverse community of developers from many different organizations. If you or your organization are trying Spark for the first time, you might be interested in the history of the project. Spark started in 2009 as a research project in the UC Berkeley RAD Lab, later to become the AMPLab. The researchers in the lab had previously been working on Hadoop MapReduce, and observed that MapReduce was inefficient for iterative and interactive computing jobs. Thus, from the beginning, Spark was designed to be fast for interactive queries and iterative algorithms, bringing in ideas like support for in-memory storage and efficient fault recovery.

Research papers were published about Spark at academic conferences and soon after its creation in 2009, it was already 10—20x faster than MapReduce for certain jobs.

Some of Spark's first users were other groups inside of UC Berkeley, including machine learning researchers such as the the Mobile Millennium project, which used Spark to monitor and predict traffic congestion in the San Francisco bay Area. In a very short

time, however, many external organizations began using Spark, and today, over 50 organizations list themselves on the Spark PoweredBy page [1], and dozens speak about their use cases at Spark community events such as Spark Meetups [2] and the Spark Summit [3]. Apart from UC Berkeley, major contributors to the project currently include Yahoo!, Intel and Databricks.

In 2011, the AMPLab started to develop higher-level components on Spark, such as Shark (Hive on Spark) and Spark Streaming. These and other components are often referred to as the Berkeley Data Analytics Stack (BDAS) [4]. BDAS includes both components of Spark and other software projects that complement it, such as the Tachyon memory manager.

Spark was first open sourced in March 2010, and was transferred to the Apache Software Foundation in June 2013, where it is now a top-level project.

Spark Versions and Releases

Since its creation Spark has been a very active project and community, with the number of contributors growing with each release. Spark 1.0 had over 100 individual contributors. Though the level of activity has rapidly grown, the community continues to release updated versions of Spark on a regular schedule. Spark 1.0 was released in May 2014. This book focuses primarily on Spark 1.1.0 and beyond, though most of the concepts and examples also work in earlier versions.

Persistence layers for Spark

Spark can create distributed datasets from any file stored in the Hadoop distributed file system (HDFS) or other storage systems supported by the Hadoop APIs (including your local file system, Amazon S3, Cassandra, Hive, HBase, etc). Its important to remember that Spark does not require Hadoop, it simply has support for storage systems implementing the Hadoop APIs. Spark supports text files, SequenceFiles, Avro, Parquet, and any other Hadoop InputFormat. We will look at interacting with these data sources in the loading and saving chapter.

1. *https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark*
2. *http://www.meetup.com/spark-users/*
3. *http://spark-summit.org*
4. *https://amplab.cs.berkeley.edu/software*

CHAPTER 2

Downloading and Getting Started

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

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

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

Downloading Spark

The first step to using Spark is to download and unpack it into a usable form. Let's start by downloading a recent precompiled released version of Spark. Visit *http://spark.apache.org/downloads.html*, then select the package type of "Pre-built package for Hadoop 2.4", click "direct file download". This will download a compressed tar file, or "tarball," called `spark-1.1.0-bin-hadoop2.4.tgz`.

You don't need to have Hadoop, but if you have an existing Hadoop cluster or HDFS install download the matching version. Those are also available from *http://spark.apache.org/downloads.html*, by selecting a different package type, but will have slightly different file names. Building from source is also possible, and you can find the latest source code on GitHub at *http://github.com/apache/spark* or select the package type of "Source code" when downloading.

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

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

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

```
cd ~
tar -xf spark-1.1.0-bin-hadoop2.4.tgz
cd spark-1.1.0-bin-hadoop2.4
ls
```

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

- `README.md` - Contains short instructions for getting started with Spark.
- `bin` - Contains executable files that can be used to interact with Spark in various ways, e.g. the spark-shell, which we will cover later in this chapter, is in here.
- `core`, `streaming`, `python` - source code of major components of the Spark project.
- `examples` - contains some helpful Spark standalone jobs that you can look at and run to learn about the Spark API.

Don't worry about the large number of directories and files the Spark project comes with; we will cover most of these in the rest of this book. For now, let's dive in right away and try out Spark's Python and Scala shells. We will start by running some of the examples that come with Spark. Then we will write, compile and run a simple Spark Job of our own.

All of the work we will do in this chapter will be with Spark running in "local mode", i.e. non-distributed mode, which only uses a single machine. Spark can run in a variety

of different modes, or environments. Beyond local mode, Spark can also be run on Mesos, YARN, on top of a Standalone Scheduler that is included in the Spark distribution. We will cover the various deployment modes in detail in chapter (to come).

Introduction to Spark's Python and Scala Shells

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

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

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

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

The easiest way to demonstrate the power of Spark's shells is to start using one of them for some simple data analysis. Let's walk through the example from the Quick Start Guide in the official Spark documentation [1].

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

```
bin/pyspark
```

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

```
bin/spark-shell
```

The shell prompt should appear within a few seconds. When the shell starts, you will notice a lot of log messages. You may need to hit [Enter] once to clear the log output,

1. *http://spark.apache.org/docs/latest/quick-start.html*

and get to a shell prompt. Figure Figure 2-1 shows what the PySpark shell looks like when you open it.

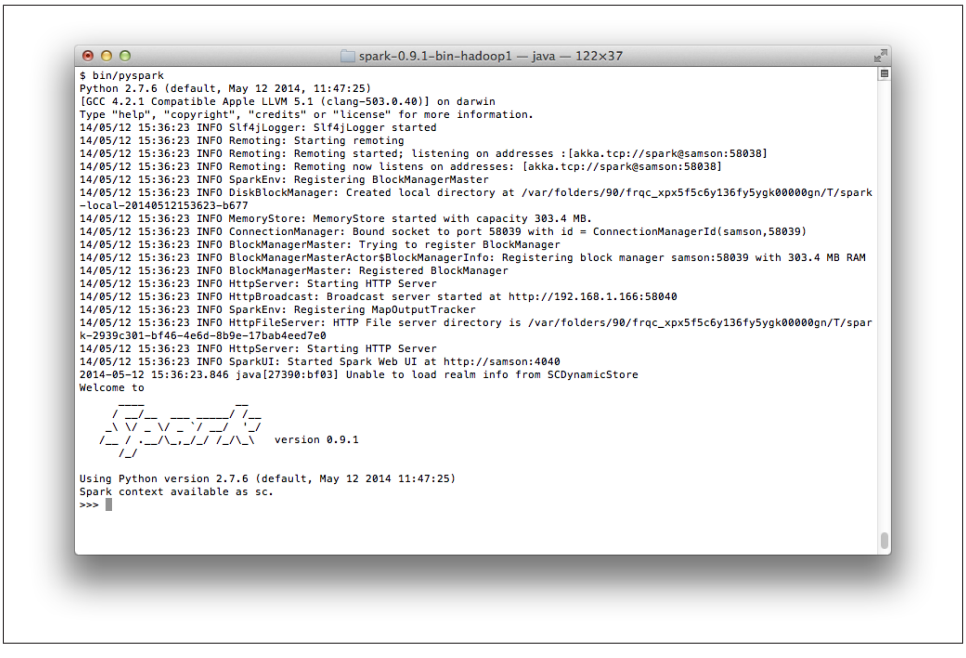

Figure 2-1. The PySpark Shell With Default Logging Output

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

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

Then lower the log level so that we only show WARN message and above by changing it to the following:

```
log4j.rootCategory=WARN, console
```

When you re-open the shell, you should see less output.

Spark will attempt to use native compression libraries if they are available, and will output a warning message if they are not. We cover compression details in Chapter 5

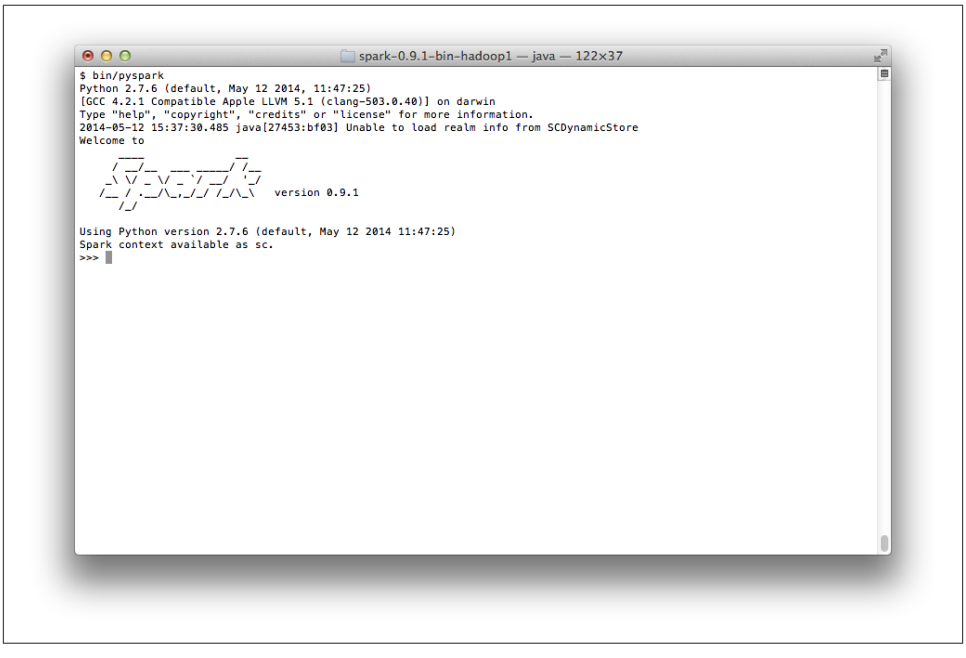

Figure 2-2. The PySpark Shell With Less Logging Output

Using IPython

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

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

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

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

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

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

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

Before we say more about RDDs, let's create one in the shell from a local text file and do some very simple ad-hoc analysis by following the example below.

Example 2-1. Python line count

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

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

Example 2-2. Scala line count

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

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

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

To exit either shell, you can press Control+D.

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

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

Introduction to Core Spark Concepts

Now that you have run your first Spark code using the shell, it's time learn about programming in it in more detail.

At a high level, every Spark application consists of a *driver program* that launches various parallel operations on a cluster. The driver program contains your application's `main` function and defines distributed datasets on the cluster, then applies operations to them. In the examples above, the driver program was the Spark shell itself, and you could just type in the operations you wanted to run.

Driver programs access Spark through a `SparkContext` object, which represents a connection to a computing cluster. In the shell, a SparkContext is automatically created for you, as the variable called `sc`. Try printing out `sc` to see its type:

```
>>> sc
<pyspark.context.SparkContext object at 0x1025b8f90>
```

Once you have a SparkContext, you can use it to build *resilient distributed datasets*, or *RDDs*. In the example above, we called `sc.textFile` to create an RDD representing the lines of text in a file. We can then run various operations on these lines, such as `count()`.

To run these operations, driver programs typically manage a number of nodes called *executors*. For example, if we were running the `count()` above on a cluster, different machines might count lines in different ranges of the file. Because we just ran the Spark shell locally, it executed all its work on a single machine — but you can connect the same shell to a cluster to analyze data in parallel. Figure 2-3 shows how Spark executes on a cluster.

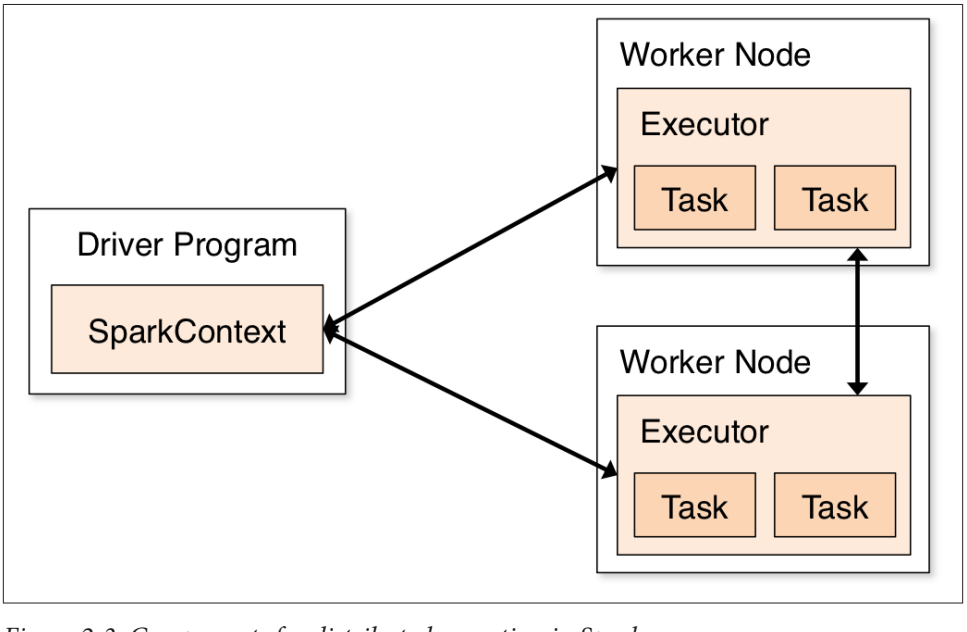

Figure 2-3. Components for distributed execution in Spark

Finally, a lot of Spark's API revolves around passing functions to its operators to run them on the cluster. For example, we could extend our README example by *filtering* the lines in the file that contain a word, such as "Python":

Example 2-3. Python filtering example

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

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

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

Example 2-4. Scala filtering example

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

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

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

If you are unfamiliar with the `lambda` or `=>` syntax above, it is a shorthand way to define functions inline in Python and Scala. When using Spark in these languages, you can also define a function separately and then pass its name to Spark. For example, in Python:

```
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```

Passing functions to Spark is also possible in Java, but in this case they are defined as classes, implementing an interface called `Function`. For example:

```
JavaRDD<String> pythonLines = lines.filter(
  new Function<String, Boolean>() {
    Boolean call(String line) { return line.contains("Python"); }
  }
);
```

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

```
JavaRDD<String> pythonLines = lines.filter(line -> line.contains("Python"));
```

We discuss passing functions further in "Passing Functions to Spark" on page 29.

While we will cover the Spark API in more detail later, a lot of its magic is that function-based operations like `filter` *also* parallelize across the cluster. That is, Spark automatically takes your function (e.g. `line.contains("Python")`) and ships it to executor

nodes. Thus, you can write code in a single driver program and automatically have parts of it run on multiple nodes. Chapter 3 covers the RDD API in more detail.

Standalone Applications

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

The process of linking to Spark varies by language. In Java and Scala, you give your application a Maven dependency on the `spark-core` artifact published by Apache. As of the time of writing, the latest Spark version is 1.1.0, and the Maven coordinates for that are:

Example 2-5. Maven coordinates for spark-core

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.1.0
```

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

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

Example 2-6. run a python example

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

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

In Spark versions before 1.0, use `bin/pyspark my_script.py` to run Python applications instead.

Initializing a SparkContext

Once you have linked an application to Spark, you need to import the Spark packages in your program and create a SparkContext. This is done by first creating a `SparkConf` object to configure your application, and then building a SparkContext for it. Here is a short example in each supported language:

Example 2-7. Initializing Spark in Python

```
from pyspark import SparkConf, SparkContext

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

Example 2-8. Initializing Spark in Scala

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

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

Example 2-9. Initializing Spark in Java

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

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

These examples show the minimal way to initialize a SparkContext, where you pass two parameters:

- A *cluster URL*, namely "local" in these examples, which tells Spark how to connect to a cluster. "local" is a special value that runs Spark on one thread on the local machine, without connecting to a cluster.
- An *application name*, namely "My App" in these examples. This will identify your application on the cluster manager's UI if you connect to a cluster.

Additional parameters exist for configuring how your application executes or adding code to be shipped to the cluster, but we will cover these in later chapters of the book.

After you have initialized a SparkContext, you can use all the methods we showed before to create RDDs (e.g. from a text file) and manipulate them.

Finally, to shut down Spark, you can either call the `stop()` method on your SparkContext, or simply exit the application (e.g. with `System.exit(0)` or `sys.exit()`).

This quick overview should be enough to let you run a standalone Spark application on your laptop. For more advanced configuration, a later chapter in the book will cover how to connect your application to a cluster, including packaging your application so that its code is automatically shipped to worker nodes. For now, please refer to the Quick Start Guide [2] in the official Spark documentation.

Building standalone applications

This wouldn't be a complete introduction chapter of a Big Data book if we didn't have a word count example. On a single machine implementing word count is quite simple, but it is a popular Big Data framework example as in a distributed mode it involves reading data from potentially many machines and combining the results from many worker nodes. We can look at building and packaging the simple WordCount example with both sbt and maven. All of our examples are able to be built together, but to illustrate a simple stripped down build with minimal dependencies we have a separate smaller project underneath the `learning-spark-examples/mini-complete-example` directory.

Example 2-10. Simple WordCount Java, don't worry about the details yet

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

Example 2-11. Simple WordCount Scala example, don't worry about the details yet

```
// Create a Scala Spark Context.
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
```

2. *http://spark.apache.org/docs/latest/quick-start.html*

```
// Load our input data.
val input =  sc.textFile(inputFile)
// Split it up into words.
val words = input.flatMap(line => line.split(" "))
// Transform into word and count.
val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y}
// Save the word count back out to a text file, causing evaluation.
counts.saveAsTextFile(outputFile)
```

We can build these using a very simple build files with both sbt and maven

Example 2-12. sbt build

build.sbt

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

version := "0.0.1"

scalaVersion := "2.10.4"

// additional libraries
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.1.0" % "provided"
)
```

Example 2-13. maven build example

```
<project>
  <groupId>com.oreilly.learningsparkexamples.mini</groupId>
  <artifactId>learning-spark-mini-example</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>example</name>
  <packaging>jar</packaging>
  <version>0.0.1</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <properties>
    <java.version>1.6</java.version>
  </properties>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.1</version>
```

```
          <configuration>
            <source>${java.version}</source>
            <target>${java.version}</target>
          </configuration>
        </plugin>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```

The spark core package is marked as provided for if we make an assembly jar. This is covered in more detail in (to come).

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

Example 2-14. scala build and run example

```
./sbt/sbt clean compile package
$SPARK_HOME/bin/spark-submit --class com.oreilly.learningsparkexamples.mini.scala.WordCount \
 ./target/scala-2.10/learning-spark-mini-example-0.0.1.jar ./README.md ./wordcounts
```

Example 2-15. maven build and run example

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

For even more detailed examples of linking applications to Spark, refer to the Quick Start Guide [3] in the official Spark documentation. Chapter 7 covers packaging Spark applications in more detail.

Conclusion

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

3. *http://spark.apache.org/docs/latest/quick-start.html*

CHAPTER 3
Programming with RDDs

This chapter introduces Spark's core abstraction for working with data, the Resilient Distributed Dataset (RDD). An RDD is simply a distributed collection of elements. In Spark all work is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result. Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them.

Both Data Scientists and Engineers should read this chapter, as RDDs are the core concept in Spark. We highly recommend that you try some of these examples in an interactive shell (see "Introduction to Spark's Python and Scala Shells" on page 11). In addition, all code in this chapter is available in the book's GitHub repository.

RDD Basics

An RDD in Spark is simply an immutable distributed collection of objects. Each RDD is split into multiple *partitions*, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java or Scala objects, including user-defined classes.

Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g. a list or set) in their driver program. We have already seen loading a text file as an RDD of strings using `SparkContext.textFile()`:

Example 3-1. Creating an RDD of strings with `textFile()` in Python

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

Once created, RDDs offer two types of operations: *transformations* and *actions*. *Transformations* construct a new RDD from a previous one. For example, one transformation

we saw before is filtering data that matches a predicate. In our text file example, we can use this to create a new RDD holding just the strings that contain "Python":

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

Actions, on the other hand, compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS). One example of an action we called earlier is `first()`, which returns the first element in an RDD:

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

The difference between transformations and actions is due to the way Spark computes RDDs. Although you can define new RDDs any time, Spark only computes them in a *lazy* fashion, the first time they are used in an action. This approach might seem unusual at first, but makes a lot of sense when working with big data. For instance, consider the example above, where we defined a text file and then filtered the lines with "Python". If Spark were to load and store all the lines in the file as soon as we wrote `lines = sc.textFile(...)`, it would waste a lot of storage space, given that we then immediately filter out many lines. Instead, once Spark sees the whole chain of transformations, it can compute just the data needed for its result. In fact, for the `first()` action, Spark only scans the file until it finds the first matching line; it doesn't even read the whole file.

Finally, Spark's RDDs are by default recomputed each time you run an action on them. If you would like to reuse an RDD in multiple actions, you can ask Spark to *persist* it using `RDD.persist()`. We can ask Spark to persist our data in a number of different places covered in Table 3-6. After computing it the first time, Spark will store the RDD contents in memory (partitioned across the machines in your cluster), and reuse them in future actions. Persisting RDDs on disk instead of memory is also possible. The behavior of not persisting by default may again seem unusual, but it makes a lot of sense for big datasets: if you will not reuse the RDD, there's no reason to waste storage space when Spark could instead stream through the data once and just compute the result.[1]

In practice, you will often use `persist` to load a subset of your data into memory and query it repeatedly. For example, if we knew that we wanted to compute multiple results about the README lines that contain "Python", we could write:

```
>>> pythonLines.persist(StorageLevel.MEMORY_ONLY_SER)

>>> pythonLines.count()
2

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

1. The ability to always recompute an RDD is actually why RDDs are called "resilient". When a machine holding RDD data fails, Spark uses this ability to recompute the missing partitions, transparent to the user.

To summarize, every Spark program and shell session will work as follows:

1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations like `filter()`.
3. Ask Spark to `persist()` any intermediate RDDs that will need to be reused.
4. Launch actions such as `count()` and `first()` to kick off a parallel computation, which is then optimized and executed by Spark.

`cache()` is the same as calling `persist()` with the default storage level.

In the rest of this chapter, we'll go through each of these steps in detail, and cover some of the most common RDD operations in Spark.

Creating RDDs

Spark provides two ways to create RDDs: loading an external dataset and parallelizing a collection in your driver program.

The simplest way to create RDDs is to take an existing in-memory collection and pass it to SparkContext's `parallelize` method. This approach is very useful when learning Spark, since you can quickly create your own RDDs in the shell and perform operations on them. Keep in mind however, that outside of prototyping and testing, this is not widely used since it requires you have your entire dataset in memory on one machine.

Example 3-2. Python `parallelize` example

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

Example 3-3. Scala `parallelize` example

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

Example 3-4. Java `parallelize` example

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "i like pandas"));
```

A more common way to create RDDs is to load data from external storage. Loading external datasets is covered in detail in Chapter 5. However, we already saw one method that loads a text file as an RDD of strings, `SparkContext.textFile`:

Example 3-5. Python `textFile` example

```
lines = sc.textFile("/path/to/README.md")
```

Example 3-6. Scala `textFile` example

```
val lines = sc.textFile("/path/to/README.md")
```

Example 3-7. Java `textFile` example

```
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

RDD Operations

RDDs support two types of operations, *transformations* and *actions*. Transformations are operations on RDDs that return a new RDD, such as `map` and `filter`. Actions are operations that return a result back to the driver program or write it to storage, and kick off a computation, such as `count` and `first`. Spark treats transformations and actions very differently, so understanding which type of operation you are performing will be important. If you are ever confused whether a given function is a transformation or an action, you can look at its return type: transformations return RDDs whereas actions return some other data type.

Transformations

Transformations are operations on RDDs that return a new RDD. As discussed shortly in the lazy evaluation section, transformed RDDs are computed lazily, only when you use them in an action. Many transformations are element-wise, that is they work on one element at a time, but this is not true for all transformations.

As an example, suppose that we have a log file, `log.txt`, with a number of messages, and we want to select only the error messages. We can use the `filter` transformation seen before. This time though, we'll show a filter in all three of Spark's language APIs:

Example 3-8. Python filter example

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

Example 3-9. Scala filter example

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

Example 3-10. Java filter example

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
  new Function<String, Boolean>() {
```

```
    public Boolean call(String x) { return x.contains("error");
  }
});
```

Note that the `filter` operation does not mutate the existing `inputRDD`. Instead, it returns a pointer to an entirely new RDD. `inputRDD` can still be re-used later in the program, for instance, to search for other words. In fact, let's use `inputRDD` again to search for lines with the word "warning" in them. Then, we'll use another transformation, `union`, to print out the number of lines that contained either "error" or "warning". We show Python here, but the `union()` function is identical in all three languages:

Example 3-11. Python union example

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

`union` is a bit different than `filter`, in that it operates on two RDDs instead of one. Transformations can actually operate on any number of input RDDs.

Finally, as you derive new RDDs from each other using transformations, Spark keeps track of the set of dependencies between different RDDs, called the *lineage graph*. It uses this information to compute each RDD on demand and to recover lost data if part of a persistent RDD is lost. We will show a lineage graph for this example in Figure 3-1.

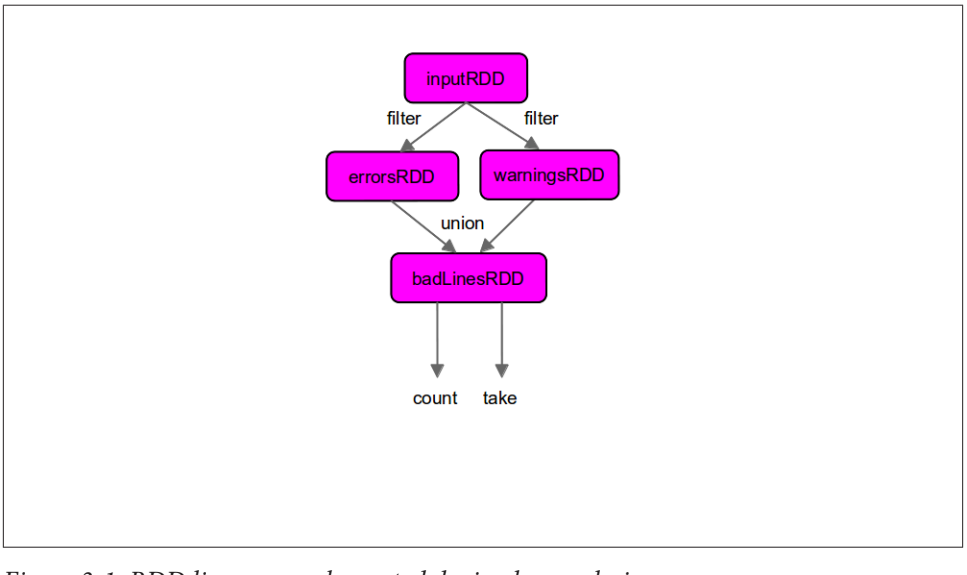

Figure 3-1. RDD lineage graph created during log analysis.

Actions

We've seen how to create RDDs from each other with transformations, but at some point, we'll want to actually *do something* with our dataset. Actions are the second type of RDD operation. They are the operations that return a final value to the driver program or write data to an external storage system. Actions force the evaluation of the transformations required for the RDD they are called on, since they are required to actually produce output.

Continuing the log example from the previous section, we might want to print out some information about the `badLinesRDD`. To do that, we'll use two actions, `count()`, which returns the count as a number, and `take()`, which collects a number of elements from the RDD.

Example 3-12. Python error count example using actions

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

Example 3-13. Scala error count example using actions

```
println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

Example 3-14. Java error count example using actions

```
System.out.println("Input had " + badLinesRDD.count() + " concerning lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
  System.out.println(line);
}
```

In this example, we used `take()` to retrieve a small number of elements in the RDD at the driver program. We then iterate over them locally to print out information at the driver. RDDs also have a `collect()` function to retrieve the entire RDD. This can be useful if your program filters RDDs down to a very small size and you'd like to deal with it locally. Keep in mind that your entire dataset must fit in memory on a single machine to use `collect()` on it, so `collect()` shouldn't be used on large datasets.

In most cases RDDs can't just be `collect()`'ed to the driver because they are too large. In these cases, it's common to write data out to a distributed storage systems such as HDFS or Amazon S3. The contents of an RDD can be saved using the `saveAsTextFile` action, `saveAsSequenceFile` or any of a number actions for various built-in formats. We will cover the different options for exporting data later on in Chapter 5.

The image below presents the lineage graph for this entire example, starting with our `inputRDD` and ending with the two actions. It is important to note that each time we call a new action, the entire RDD must be computed "from scratch". To avoid this inefficiency, users can *persist* intermediate results, as we will cover in "Persistence (Caching)" on page 44.

Lazy Evaluation

Transformations on RDDs are lazily evaluated, meaning that Spark will not begin to execute until it sees an action. This can be somewhat counter-intuitive for new users, but may be familiar for those who have used functional languages such as Haskell or LINQ-like data processing frameworks.

Lazy evaluation means that when we call a transformation on an RDD (for instance calling `map`), the operation is not immediately performed. Instead, Spark internally records meta-data to indicate this operation has been requested. Rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through transformations. Loading data into an RDD is lazily evaluated in the same way transformations are. So when we call `sc.textFile` the data is not loaded until it is necessary. Like with transformations, the operation (in this case reading the data) can occur multiple times.

Although transformations are lazy, we can force Spark to execute them at any time by running an action, such as `count()`. This is an easy way to test out just part of your program.

Spark uses lazy evaluation to reduce the number of passes it has to take over our data by grouping operations together. In systems like Hadoop MapReduce, developers often have to spend a lot time considering how to group together operations to minimize the number of MapReduce passes. In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations. Thus, users are free to organize their program into smaller, more manageable operations.

Passing Functions to Spark

Most of Spark transformations, and some of its actions, depend on passing in functions that are used by Spark to compute data. Each of the core languages has a slightly different mechanism for passing functions to Spark.

Python

In Python, we have three options for passing functions into Spark. For shorter functions we can pass in `lambda` expressions, as we have done in the example at the start of this chapter. We can also pass in top-level functions, or locally defined functions.

Example 3-15. Passing a lambda in Python

```
word = rdd.filter(lambda s: "error" in s)
```

Passing a top-level Python function.

```
def containsError(s):
    return "error" in s
word = rdd.filter(containsError)
```

One issue to watch out for when passing functions is inadvertently serializing the object containing the function. When passing a function that is the member of an object, or contains references to fields in an object (e.g., `self.field`), Spark sends the *entire* object, which can be much larger than just the bit of information you need. Sometimes this can also cause your program to fail, if your class contains objects that Python can't figure out how to pickle.

Example 3-16. Passing a function with field references (don't do this!)

```
class SearchFunctions(object):
  def __init__(self, query):
      self.query = query
  def isMatch(self, s):
      return self.query in s
  def getMatchesFunctionReference(self, rdd):
      # Problem: references all of "self" in "self.isMatch"
      return rdd.filter(self.isMatch)
  def getMatchesMemberReference(self, rdd):
      # Problem: references all of "self" in "self.query"
      return rdd.filter(lambda x: self.query in x)
```

Instead, just extract the fields you need from your object into local variable and pass that in, like we do below:

Example 3-17. Python function passing without field references

```
class WordFunctions(object):
  ...
  def getMatchesNoReference(self, rdd):
      # Safe: extract only the field we need into a local variable
      query = self.query
      return rdd.filter(lambda x: query in x)
```

Scala

In Scala, we can pass in functions defined inline, references to methods, or static functions as we do for Scala's other functional APIs. Some other considerations come into play though, namely that the function we pass and the data referenced in it needs to be Serializable (implementing Java's Serializable interface). Furthermore, like in Python, passing a method or field of an object includes a reference to that whole object, though this is less obvious because we are not forced to write these references with `self`. Like how we did with Python, we can instead extract out the fields we need as local variables and avoid needing to pass the whole object containing them.

Example 3-18. Scala function passing

```
class SearchFunctions(val query: String) {
  def isMatch(s: String): Boolean = {
    s.contains(query)
  }
  def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {
    // Problem: "isMatch" means "this.isMatch", so we pass all of "this"
    rdd.map(isMatch)
  }
  def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {
    // Problem: "query" means "this.query", so we pass all of "this"
    rdd.map(x => x.split(query))
  }
  def getMatchesNoReference(rdd: RDD[String]): RDD[String] = {
    // Safe: extract just the field we need into a local variable
    val query_ = this.query
    rdd.map(x => x.split(query_))
  }
}
```

If a "NotSerializableException" exception errors in Scala, a reference to a method or field in a non-serializable class is usually the problem. Note that passing in local serializable variables or functions that are members of a top-level `object` is always safe.

Java

In Java, functions are specified as objects that implement one of Spark's function interfaces from the `org.apache.spark.api.java.function` package. There are a number of different interfaces based on the return type of the function. We show the most basic function interfaces below, and cover a number of other function interfaces for when we need to return special types of data, like key-value data, in the section on converting between RDD types.

Table 3-1. Standard Java function interfaces

Function name	method to implement	Usage
`Function<T, R>`	`R call(T)`	Take in one input and return one output, for use with things like `map` and `filter`.
`Function2<T1, T2, R>`	`R call(T1, T2)`	Take in two inputs and return one output, for use with things like `aggregate` or `fold`.
`FlatMapFunction<T, R>`	`Iterable<R> call(T)`	Take in one input and return zero or more outputs, for use with things like `flatMap`.

We can either define our function classes in-line as anonymous inner classes, or make a named class:

Example 3-19. Java function passing with anonymous inner class

```
RDD<String> errors = lines.filter(new Function<String, Boolean>() {
  public Boolean call(String x) { return x.contains("error"); }
});
```

Example 3-20. Java function passing with named class

```
class ContainsError implements Function<String, Boolean>() {
  public Boolean call(String x) { return x.contains("error"); }
}

RDD<String> errors = lines.filter(new ContainsError());
```

The style to choose is a personal preference, but we find that top-level named functions are often cleaner for organizing large programs. One other benefit of top-level functions is that you can give them constructor parameters:

Example 3-21. Java function class with parameters

```
class Contains implements Function<String, Boolean>() {
  private String query;
  public Contains(String query) { this.query = query; }
  public Boolean call(String x) { return x.contains(query); }
}

RDD<String> errors = lines.filter(new Contains("error"));
```

In Java 8, you can also use lambda expressions to concisely implement the Function interfaces. Since Java 8 is still relatively new as of the writing of this book, our examples use the more verbose syntax for defining classes in previous versions of Java. However, with lambda expressions, our search example would look like this:

Example 3-22. Java function passing with lambda expression in Java 8

```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

If you are interested in using Java 8's lambda expression, refer to Oracle's documentation and the Databricks blog post on how to use lambdas with Spark.

Both anonymous inner classes and lambda expressions can reference any `final` variables in the method enclosing them, so you can pass these variables to Spark just like in Python and Scala.

Common Transformations and Actions

In this chapter, we tour the most common transformations and actions in Spark. Additional operations are available on RDDs containing certain type of data — for example, statistical functions on RDDs of numbers, and key-value operations such as aggregating data by key on RDDs of key-value pairs. We cover converting between RDD types and these special operations in later sections.

Basic RDDs

We will begin by evaluating what operations we can do on all RDDs regardless of the data. These transformations and actions are available on all RDD classes.

Transformations

Element-wise transformations

The two most common transformations you will likely be performing on basic RDDs are `map`, and `filter`. The `map` transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD. The `filter` transformation take in a function and returns an RDD which only has elements that pass the filter function.

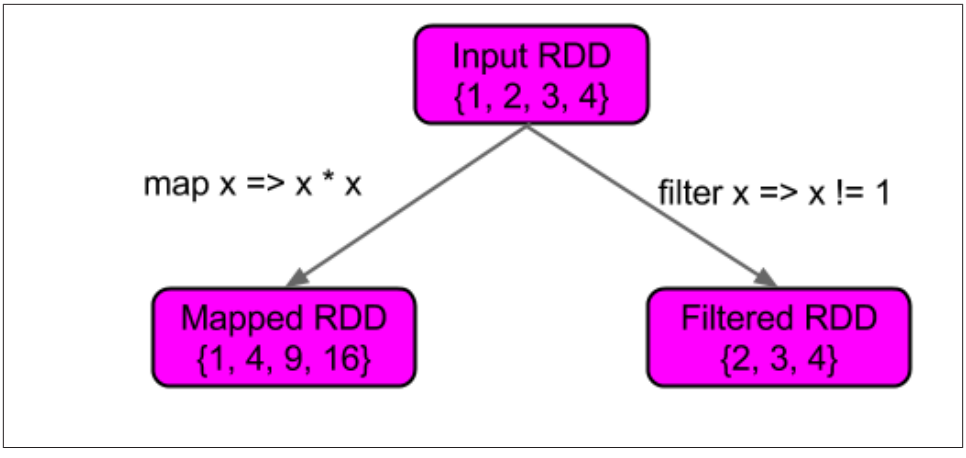

Figure 3-2. Mapped and Filtered RDD from an Input RDD

We can use `map` to do any number of things from fetching the website associated with each URL in our collection, to just squaring the numbers. With Scala and python you can use the standard anonymous function notation or pass in a function, and with Java you should use Spark's `Function` class from `org.apache.spark.api.java.function` or Java 8 functions.

It is useful to note that the return type of the `map` does not have to be the same as the input type, so if we had an RDD Strings and our map function were to parse the strings and return a Double the type of our input RDD would be `RDD[String]` and the resulting RDD would be of the type `RDD[Double]`.

Lets look at a basic example of `map` which squares all of the numbers in an RDD:

Example 3-23. Python squaring the value in an RDD

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

Example 3-24. Scala squaring the values in an RDD

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

Example 3-25. Java squaring the values in an RDD

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {
  public Integer call(Integer x) { return x*x; }
```

```
});
System.out.println(StringUtils.join(result.collect(), ","));
```

Sometimes we want to produce multiple output elements for each input element. The operation to do this is called `flatMap`. Like with `map`, the function we provide to `flatMap` is called individually for each element in our input RDD. Instead of returning a single element, we return an iterator with our return values. Rather than producing an RDD of iterators, we get back an RDD which consists of the elements from all of the iterators. A simple example of `flatMap` is splitting up an input string into words, as shown below.

Example 3-26. Python `flatMap` example, splitting lines into words

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first()  # returns "hello"
```

Example 3-27. Scala `flatMap` example, splitting lines into multiple words

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first()  // returns "hello"
```

Example 3-28. Java `flatMap` example, splitting lines into multiple words

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String line) {
    return Arrays.asList(line.split(" "));
  }
});
words.first();  // returns "hello"
```

We illustrate the different between `flatMap` and `map` in Figure 3-3 and Figure 3-4. You can think of `flatMap` as "flattening", like in Perl, the iterators returned to it so instead of ending up with an RDD of lists we have an RDD of the elements in those lists.

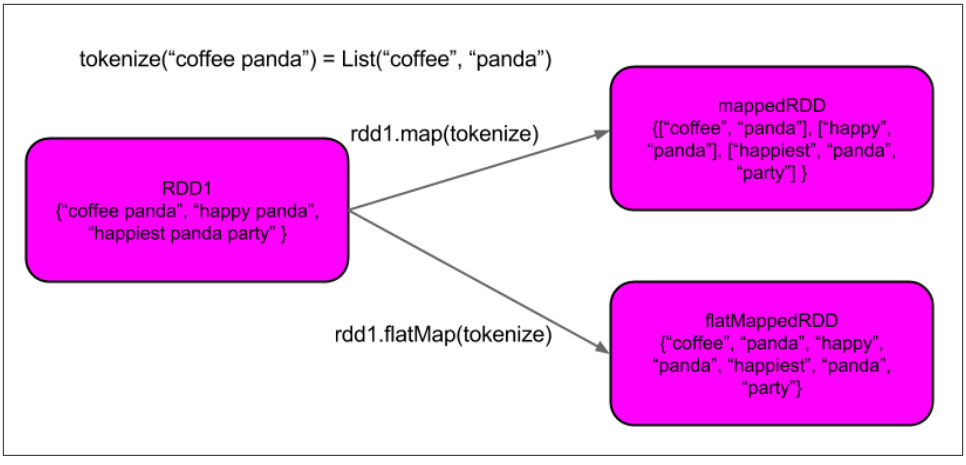

Figure 3-3. Difference between flatMap and map on an RDD

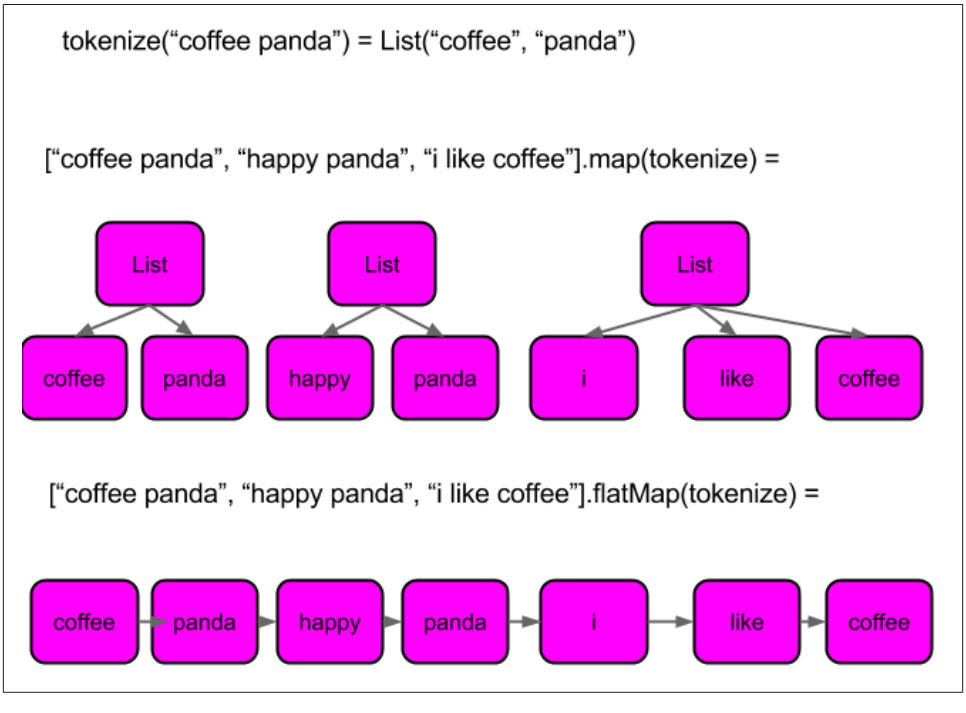

Figure 3-4. Difference between flatMap and map

Pseudo Set Operations

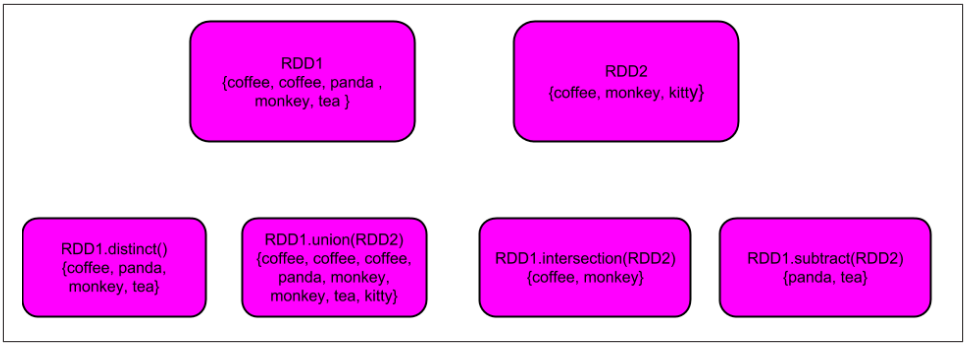

Figure 3-5. Some simple set operations (image to be redone)

RDDs support many of the operations of mathematical sets, such as union and intersection, even when the RDDs themselves are not properly sets. Its important to note that all of these operations require that the RDDs being operated on are of the same type.

The set property most frequently missing from our RDDs is the uniqueness of elements, as we often have duplicates. If we only want unique elements we can use the `RDD.distinct()` transformation to produce a new RDD with only distinct items. Note that `distinct()` is expensive, however, as it requires shuffling all the data over the network to ensure that we only receive one copy of each element. Shuffling, and how to avoid it, is discussed in more detail in Chapter 4.

The simplest set operation is `union(other)`, which gives back an RDD consisting of the data from both sources. This can be useful in a number of use cases, such as processing log files from many sources. Unlike the mathematical `union()`, if there are duplicates in the input RDDs, the result of Spark's `union()` will contain duplicates (which we can fix if desired with `distinct()`).

Spark also provides an `intersection(other)` method, which returns only elements in both RDDs. `intersection()` also removes all duplicates (including duplicates from a single RDD) while running. While `intersection` and `union` are to very similar concepts, the performance of `intersection` is much worse since it requires a shuffle over the network to identify common elements. Shuffle operations are commonly required for `joins` and involve moving data around inside the cluster so that each worker has the data required for the join or intersection.

Sometimes we need to remove some data from consideration. The `subtract(other)` function takes in another RDD and returns an RDD that only has values present in the first RDD and not the second RDD.

We can also compute a Cartesian product between two RDDs. The `cartesian(other)` transformation results in possible pairs of `(a, b)` where a is in the source RDD and

b is in the other RDD. The Cartesian product can be useful when we wish to consider the similarity between all possible pairs, such as computing every users expected interests in each offer. We can also take the Cartesian product of an RDD with itself, which can be useful for tasks like computing user similarity.

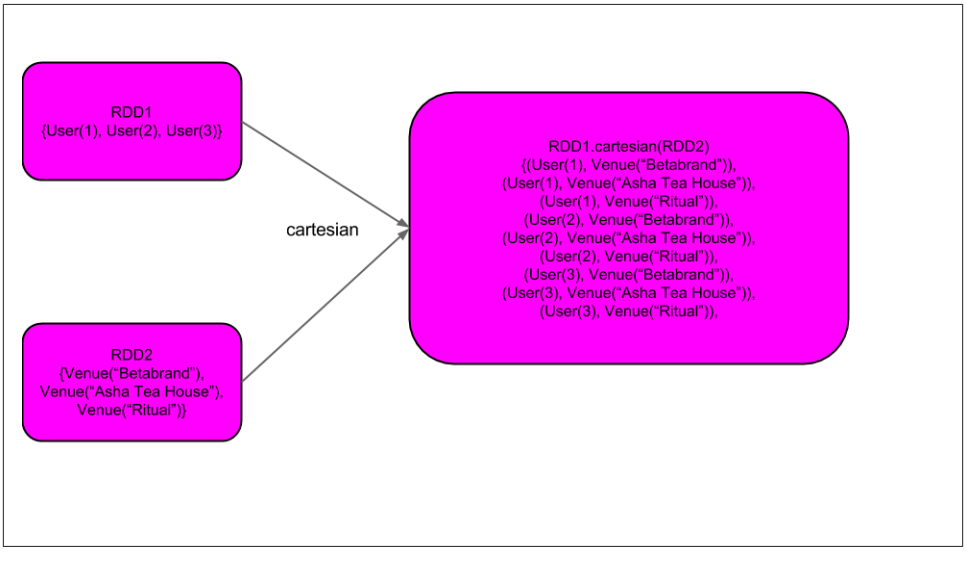

Figure 3-6. Cartesian product between two RDDs

The tables below summarize common single-RDD and multi-RDD transformations.

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function Name	Purpose	Example	Result
map	Apply a function to each element in the RDD and return an RDD of the result	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
flatMap	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
filter	Return an RDD consisting of only elements which pass the condition passed to filter	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
distinct	Remove duplicates	`rdd.distinct()`	`{1, 2, 3}`
sample(withReplacement, fraction, [seed])	Sample an RDD	`rdd.sample(false, 0.5)`	non-deterministic

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function Name	Purpose	Example	Result
union	Produce an RDD contain elements from both RDDs	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
intersection	RDD containing only elements found in both RDDs	`rdd.intersection(other)`	`{3}`
subtract	Remove the contents of one RDD (e.g. remove training data)	`rdd.subtract(other)`	`{1, 2}`
cartesian	Cartesian product with the other RDD	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

As you can see there are a wide variety of transformations available on all RDDs regardless of our specific underlying data. We can transform our data element-wise, obtain distinct elements, and do a variety of set operations.

Actions

The most common action on basic RDDs you will likely use is `reduce`. Reduce takes in a function which operates on two elements of the same type of your RDD and returns a new element of the same type. A simple example of such a function is + , which we can use to sum our RDD. With reduce we can easily sum the elements of our RDD, count the number of elements, and perform other types of aggregations.

Example 3-29. Python `reduce` example

```
sum = rdd.reduce(lambda x, y: x + y)
```

Example 3-30. Scala `reduce` example

```
val sum = rdd.reduce((x, y) => x + y)
```

Example 3-31. Java `reduce` example

```
Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer x, Integer y) { return x + y;}
});
```

Similar to `reduce` is `fold` which also takes a function with the same signature as needed for `reduce`, but also takes a "zero value" to be used for the initial call on each partition. The zero value you provide should be the identity element for your operation, that is applying it multiple times with your function should not change the value, (e.g. 0 for +, 1 for *, or an empty list for concatenation).

You can minimize object creation in `fold` by modifying and returning the first of the two parameters in-place. However, you should not modify the second parameter.

Fold and reduce both require that the return type of our result be the same type as that of the elements in the RDD we are operating over. This works well for doing things like sum, but sometimes we want to return a different type. For example when computing the running average we need keep track of both the count so far and the number of elements, requiring we return a pair. We could implement this using a map first where we transform every element into the element and the number 1 so that the reduce function can work on pairs.

The `aggregate` function frees us from the constraint of having the return be the same type as the RDD which we are working on. With `aggregate`, like `fold`, we supply an initial zero value of the type we want to return. We then supply a function to combine the elements from our RDD with the accumulator. Finally, we need to supply a second function to merge two accumulators, given that each node accumulates its own results locally.

We can use `aggregate` to compute the average of a RDD avoiding a `map` before the `fold`.

Example 3-32. Python `aggregate` example

```
sumCount = nums.aggregate((0, 0),
               (lambda x, y: (x[0] + y, x[1] + 1),
               (lambda x, y: (x[0] + y[0], x[1] + y[1])))
return sumCount[0] / float(sumCount[1])
```

Example 3-33. Scala `aggregate` example

```
val result = input.aggregate((0, 0))(
               (x, y) => (x._1 + y, x._2 + 1),
               (x, y) => (x._1 + y._1, x._2 + y._2))
val avg = result._1 / result._2.toDouble
```

Example 3-34. Java `aggregate` example

```
class AvgCount implements Serializable {
  public AvgCount(int total, int num) {
    this.total = total;
    this.num = num;
  }
  public int total;
  public int num;
  public double avg() {
    return total / (double) num;
  }
```

```
}
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
    public AvgCount call(AvgCount a, Integer x) {
      a.total += x;
      a.num += 1;
      return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total += b.total;
    a.num += b.num;
    return a;
  }
};
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

Some actions on RDDs return some or all of the data to our driver program in the form of a regular collection or value.

The simplest and most common operation that returns data to our driver program is `collect()`, which returns the entire RDDs contents. `collect` is commonly used in unit tests where the entire contents of the RDD is expected to fit in memory as it makes it easy to compare the value of our RDD with our expected result. `collect` suffers from the restriction that all of your data must fit on a single machine, as it all needs to be copied to the driver.

`take(n)` returns n elements from the RDD and attempts to minimize the number of partitions it accesses, so it may represent a biased collection. Its important to note that these operations do not return the elements in the order you might expect.

These operations are useful for unit tests and quick debugging, but may introduce bottlenecks when dealing with large amounts of data.

If there is an ordering defined on our data, we can also extract the top elements from an RDD using `top`. `top` will use the default ordering on the data, but we can supply our own comparison function to extract the top elements.

Sometimes we need a sample of our data in our driver program. The `takeSample(withReplacement, num, seed)` function allows us to take a sample of our data either with or without replacement. For more control, we can create a Sampled RDD and `collect` which we will talk about in the Sampling your data section in the Simple Optimizations chapter.

Sometimes it is useful to perform an action on all of the elements in the RDD, but we don't care about the result. A good example of this would be posting JSON to a webserver or inserting records into a database. In that case the `foreach` action lets us perform computation on each element in the RDD without having to bring it back locally.

The further standard operations on a basic RDD all behave pretty much exactly as one would imagine from their name. `count()` returns a count of the elements, and `countByValue()` returns a map of each unique value to its count. See the basic RDD actions table for more actions.

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function Name	Purpose	Example	Result
collect()	Return all elements from the RDD	`rdd.collect()`	`{1, 2, 3, 3}`
count()	Number of elements in the RDD	`rdd.count()`	4
countByValue()	Number of times each element occurs in the RDD	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
take(num)	Return num elements from the RDD	`rdd.take(2)`	`{1, 2}`
top(num)	Return the top num elements the RDD	`rdd.top(2)`	`{3, 3}`
takeOrdered(num)(ordering)	Return num elements based on providing ordering	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
takeSample(withReplacement, num, [seed])	Return num elements at random	`rdd.takeSample(false, 1)`	non-deterministic
reduce(func)	Combine the elements of the RDD together in parallel (e.g. sum)	`rdd.reduce((x, y) => x + y)`	9
fold(zero)(func)	Same as reduce but with the provided zero value	`rdd.fold(0)((x, y) => x + y)`	9

Function Name	Purpose	Example	Result
aggregate(zeroValue)(seqOp, combOp)	Similar to reduce but used to return a different type	`rdd.aggregate(0, 0)(x, y) => (y._1() + x, y._2() + 1),(x, y) => (y._1() + x._1(), y._2() + x._2()))`	`(9, 4)`
foreach(func)	Apply the provided function to each element of the RDD	`rdd.foreach(func)`	nothing

Converting Between RDD Types

We don't have to do anything special to get back the correct templated/generic type of RDD (that is, our RDD of Strings can become an RDD of Integers just by calling map with the correct function). Some functions are only available on certain types of RDDs, such as average on numeric RDDs and join on key-value pair RDDs. We will cover these special functions for numeric data in Chapter 6 and pair RDDs in Chapter 4.

In Scala and Java, these methods aren't defined on the standard RDD class, so to access this additional functionality we have to make sure we get the correct specialized class.

Scala

In Scala the conversion between such RDDs (like from `RDD[Double]` and `RDD[Numeric]` to `DoubleRDD`) is handled automatically using implicit conversions. As mentioned in standard imports, we need to add `import org.apache.spark.SparkContext._` for these conversion to work. You can see the implicit conversions listed in the object `SparkContext` in the Spark source code at `./core/src/main/scala/org/apache/spark/SparkContext.scala`. These implicits also allow for RDDs of Scala types to be written out to HDFS and similar.

Implicits, while quite powerful, can sometimes be confusing. If you call a function like say `stats()` on an RDD, you might look at the scaladocs for the RDD class and notice there is no `stats()` function. The call manages to succeed because of implicit conversions between `RDD[Numeric]` and `DoubleRDDFunctions`. When looking for functions on your RDD in Scaladoc make sure to look at functions that are available in the other RDD classes.

Java

In Java the conversion between the specialized types of RDDs is a bit more explicit. This has the benefit of giving you a greater understanding of what exactly is going on, but can be a bit more cumbersome.

To allow Spark to determine the correct return type, instead of always using the `Function` class we will need to use specialized versions. If we want to create a DoubleRDD from an RDD of type T, rather than using `Function<T, Double>` we use `DoubleFunction<T>`. The special Java functions table shows the specialized functions and their uses.

We also need to call different functions on our RDD (so we can't just create a DoubleFunction and pass it to map). When we want a DoubleRDD back instead of calling `map` we need to call `mapToDouble` with the same pattern followed with all other functions.

Table 3-5. Java interfaces for type-specific functions

Function name	Equivalent Function*<A, B,...>	Usage
`DoubleFlatMapFunction<T>`	`Function<T, Iterable<Double>>`	`DoubleRDD` from a `flatMapToDouble`
`DoubleFunction<T>`	`Function<T, double>`	`DoubleRDD` from `mapToDouble`
`PairFlatMapFunction<T, K, V>`	`Function<T, Iterable<Tuple2<K, V>>>`	`PairRDD<K, V>` from a `flatMapToPair`
`PairFunction<T, K, V>`	`Function<T, Tuple2<K, V>>`	`PairRDD<K, V>` from a `mapToPair`

We can modify our previous example where we squared an RDD of numbers to produce a `JavaDoubleRDD`. This gives us access to the additional `DoubleRDD` specific functions like `average` and `stats`.

Example 3-35. Java create DoubleRDD example

```
JavaDoubleRDD result = rdd.mapToDouble(
  new DoubleFunction<Integer>() {
    public double call(Integer x) {
      return (double) x * x;
    }
});
System.out.println(result.average());
```

Python

The Python API is structured a bit different than both the Java and Scala API. Like the Scala API, we don't need to be explicit to access the functions which are only available on Double or Pair RDDs. In Python all of the functions are implemented on the base RDD class and will simply fail at runtime if the type doesn't work.

Persistence (Caching)

As discussed earlier, Spark RDDs are lazily evaluated, and sometimes we may wish to use the same RDD multiple times. If we do this naively, Spark will recompute the RDD

and all of its dependencies each time we call an action on the RDD. This can be especially expensive for iterative algorithms, which look at the data many times. Another trivial example would be doing a count and then writing out the same RDD.

Example 3-36. Scala double execute example

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

To avoid computing an RDD multiple times, we can ask Spark to *persist* the data. When we ask Spark to persist an RDD, the nodes that compute the RDD store their partitions. If a node that has data persisted on it fails, Spark will recompute the lost partitions of the data when needed. We can also replicate our data on multiple nodes if we want to be able to handle node failure without slowdown.

Spark has many levels of persistence to chose from based on what our goals are. In Scala and Java, the default `persist()` will store the data in the JVM heap as unserialized objects. In Python, we always serialize the data that persist stores, so the default is instead stored in the JVM heap as pickled objects. When we write data out to disk or off-heap storage that data is also always serialized.

Off-heap caching is experimental and uses Tachyon. If you are interested in off-heap caching with Spark, take a look at the running Spark on Tachyon guide.

Table 3-6. Persistence levels from `org.apache.spark.storage.StorageLevel` / `pyspark.StorageLevel`. If desired we can have the data on two machines by adding `_2` to the end of the storage level.

Level	Space Used	CPU time	In memory	On Disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Example 3-37. Scala persist example

```
val result = input.map(x => x*x)
result.persist(MEMORY_ONLY)
```

```
println(result.count())
println(result.collect().mkString(","))
```

You will note that we called `persist` on the RDD before the first action. The `persist` call on its own doesn't force evaluation.

If you attempt to cache too much data to fit in memory, Spark will automatically evict old partitions using a Least Recently Used (LRU) cache policy. For the memory-only storage levels, it will recompute these partitions the next time they are accessed, while for the memory-and-disk ones, it will write them out to disk. In either case, this means that you don't have to worry about your job breaking if you ask Spark to cache too much data. However, caching unnecessary data can lead to eviction of useful data and more recomputation time.

Finally, RDDs come with a method called `unpersist()` that lets you manually remove them from the cache.

Conclusion

In this chapter, we have covered the RDD execution model and a large number of common operations on RDDs. If you have gotten here, congratulations — you've learned all the core concepts of working in Spark. In the next chapter, we'll cover a special set of operations available on RDDs of key-value pairs, which are the most common way to aggregate or group together data in parallel. After that, we discuss input and output from a variety of data sources, and more advanced topics in working with SparkContext.

CHAPTER 4

Working with Key-Value Pairs

This chapter covers how to work with RDDs of key-value pairs, which are a common data type required for many operations in Spark. Key-Value RDDs are commonly used to perform aggregations, and often we will do some initial ETL to get our data into a key-value format. Key-value RDDs expose new operations (e.g., counting up reviews for each product), grouping together data with the same key, and grouping together two different RDDs.

We also discuss an advanced feature that lets users control the layout of pair RDDs across nodes: *partitioning*. Using controllable partitioning, applications can sometimes greatly reduce communication costs, by ensuring that data that will be accessed together will be on the same node. This can provide significant speedups. We illustrate partitioning using the PageRank algorithm as an example. Choosing the right partitioning for a distributed dataset is similar to choosing the right data structure for a local one — in both cases, data layout can greatly affect performance.

Motivation

Spark provides special operations on RDDs containing key-value pairs. These RDDs are called Pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a `reduceByKey` method that can aggregate data separately for each key, and a `join` method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing for instance, an event time, customer ID, or other identifier) and use those fields as keys in Pair RDD operations.

Creating Pair RDDs

There are a number of ways to get Pair RDDs in Spark. Many formats we explore in Chapter 5 will directly return Pair RDDs for their key-value data. In other cases we have a regular RDD that we want to turn into a Pair RDD. To illustrate creating a Pair RDDs we will key our data by the first word in each line of the input.

In Python, for the functions on keyed data to work we need to make sure our RDD consists of tuples.

Example 4-1. Python create pair RDD using the first word as the key

```
input.map(lambda x: (x.split(" ")[0], x))
```

In Scala, to create Pair RDDs from a regular RDD, we simply need to return a tuple from our function.

Example 4-2. Scala create pair RDD using the first word as the key

```
input.map(x => (x.split(" ")(0), x))
```

Java doesn't have a built-in tuple type, so Spark's Java API has users create tuples using the `scala.Tuple2` class. This class is very simple: Java users can construct a new tuple by writing `new Tuple2(elem1, elem2)` and can then access the elements with the `._1()` and `._2()` methods.

Java users also need to call special versions of Spark's functions when creating Pair RDDs. For instance, the `mapToPair` function should be used in place of the basic `map` function. This is discussed in more detail in converting between RDD types, but lets look at a simple example below.

Example 4-3. Java create pair RDD using the first word as the key

```
PairFunction<String, String, String> keyData =
  new PairFunction<String, String, String>() {
  public Tuple2<String, String> call(String x) {
    return new Tuple2(x.split(" ")[0], x);
  }
};
JavaPairRDD<String, String> rdd = input.mapToPair(keyData);
```

When creating a Pair RDD from an in memory collection in Scala and Python we only need to make sure the types of our data are correct, and call `parallelize`. To create a Pair RDD in Java from an in memory collection we need to make sure our collection consists of tuples and also call `SparkContext.parallelizePairs` instead of `SparkContext.parallelize`.

Transformations on Pair RDDs

Pair RDDs are allowed to use all the transformations available to standard RDDs. The same rules from passing functions to spark apply. Since Pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements. The following tables, Table 4-1 and Table 4-2, summarize transformations on pair RDDs and we will dive into the transformations in detail later on in the chapter.

Table 4-1. Transformations on one Pair RDD (example ({(1, 2), (3, 4), (3, 6)}))

Function Name	Purpose	Example	Result
combineByKey(`createCombiner`, `mergeValue`, `mergeCombiners`, `partitioner`)	Combine values with the same key together	See combine by key example	
`groupByKey()`	Group together values with the same key	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`reduceByKey(func)`	Combine values with the same key together	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`mapValues(func)`	Apply a function to each value of a Pair RDD without changing the key	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function which returns an iterator to each value of a Pair RDD and for each element returned produce a key-value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => x.to(5))`	`{(1,2), (1,3), (1,4), (1,5), (3, 4), (3,5)}`
`keys()`	Return an RDD of just the keys	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of just the values	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Returns an RDD sorted by the key	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`

Table 4-2. Transformations on two Pair RDD (example ({(1, 2), (3, 4), (3, 6)}) other ({(3, 9)})

Function Name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD	`rdd.subtractByKey(other)`	`{1, 2}`
`join`	Perform an inner join between two RDDs	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDS where the key must be present in the first RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDS where the key must be present in the other RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group together data from both RDDs sharing the same key	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

In Java and Scala when we run a `map` or `filter` or similar over a Pair RDDs, our function will be passed an instance of `scala.Tuple2`. In Scala pattern matching is a common way of extracting the individual values, whereas in Java we use the `._1()` and `._2()` values to access the elements. In Python our Pair RDDs consist of standard Python tuple objects that we interact with as normal.

For instance, we can take our Pair RDD from the previous section and filter out lines longer than 20 characters.

Example 4-4. Python simple filter on second element

```
result = pair.filter(lambda x: len(x[1]) < 20)
```

Example 4-5. Scala simple filter on second element

```
pair.filter{case (x, y) => y.length < 20}
```

Example 4-6. Java simple filter on second element

```
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> input) {
      return (input._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result = rdd.filter(longWordFilter);
```

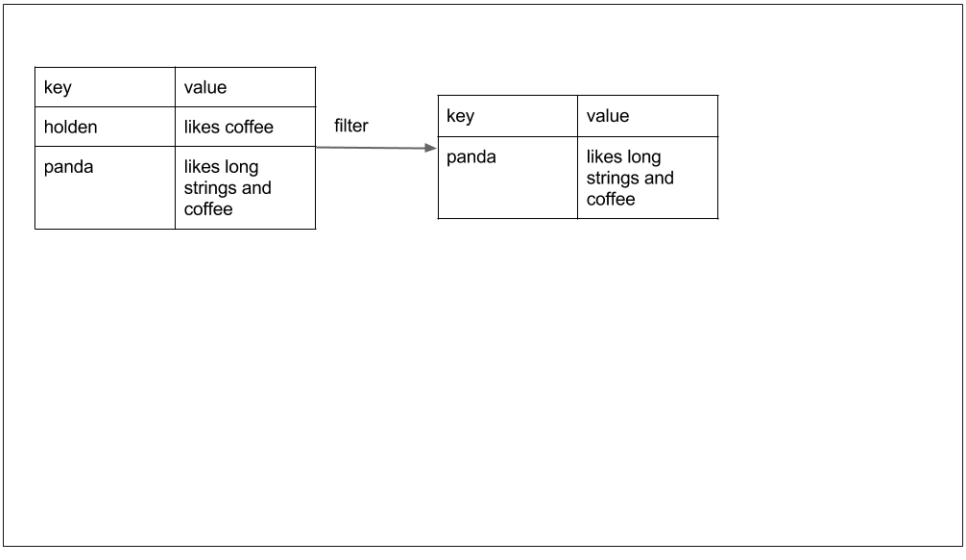

Figure 4-1. Filter on value

Sometimes working with these pairs can be awkward if we only want to access the value part of our Pair RDD. Since this is a common pattern, Spark provides the `mapValues(func)` function which is the same as `map{case (x, y) (x, func(y)}` and we will use this function in many of our examples.

Aggregations

When datasets are described in terms of key-value pairs, it is common to want to aggregate statistics across all elements with the same key. We have looked at the `fold`, `combine`, and `reduce` actions on basic RDDs, and similar per-key transformations exist on Pair RDDs. Spark has a similar set of operations which combine the values together which have the same key. Instead of being actions these operations return RDDs and as such are transformations.

`reduceByKey` is quite similar to `reduce`, both take a function and use it to combine values. `reduceByKey` runs several parallel reduce operations, one for each key in the dataset, where each operation combines values together which have the same key. Because datasets can have very large numbers of keys, `reduceByKey` is not implemented as an action that returns a value back to the user program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.

`foldByKey` is quite similar to `fold`, both use a zero value of the same type of the data in our RDD and combination function. Like with `fold` the provided zero value for `fold`

ByKey should have no impact when added with your combination function to another element.

We can use reduceByKey along with mapValues to compute the per-key average in a very similar manner to how we used fold and map compute the entire RDD average. As with averaging, we can achieve the same result using a more specialized function we will cover next.

Example 4-7. Python per key average with reduceByKey and mapValues

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

Example 4-8. Scala per key average with reduceByKey and mapValues

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2))
```

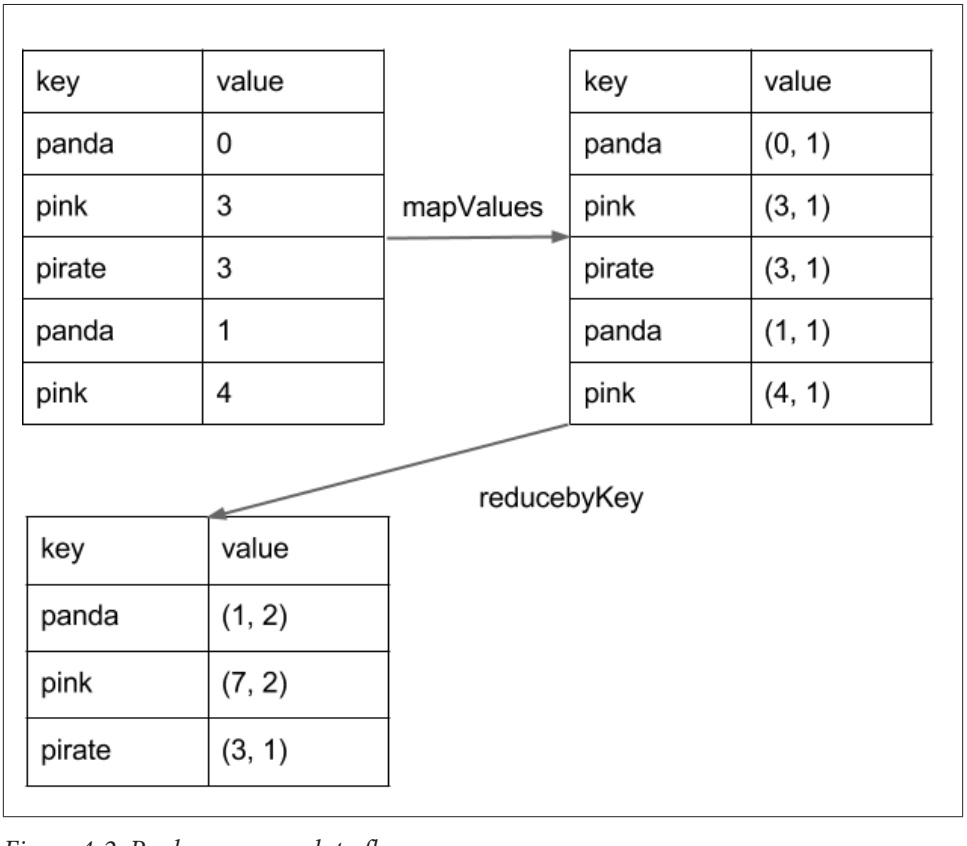

Figure 4-2. Per key average data flow

Those familiar with the combiner concept from MapReduce should note that calling `reduceByKey` and `foldByKey` will automatically perform combining locally on each machine before computing global totals for each key. The user does not need to specify a combiner. The more general `combineByKey` interface allows you to customize combining behavior.

We can use a similar approach to also implement the classic distributed word count problem. We will use `flatMap` from the previous chapter so that we can produce a Pair RDD of words and the number 1 and then sum together all of the words using `reduceByKey` like in our previous example.

Example 4-9. Python word count example

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

Example 4-10. Scala word count example

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

Example 4-11. Java word count example

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
}).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
```

We can actually implement word count even faster by using the `countByValue()` function on the first RDD: `input.flatMap(x => x.split(" ")).countByValue()`.

`combineByKey` is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it. Like `aggregate`, `combineByKey` allows the user to return values which are not the same type as our input data.

To understand `combineByKey` its useful to think of how it handles each element it processes. As `combineByKey` goes through the elements in a partition, each element either has a key it hasn't seen before or has the same key as a previous element. If its a new element then `combineByKey` uses a function we provide, called `createCombiner`, to create the initial value for the accumulator on that key. It important to note that this happens the first time a key is found in each partition, rather than only the first time the key is found in the RDD. If it is a value the we have seen before while processing that partition it will instead use the provided function, `mergeValue`, with the current value for the accumulator for that key and the new value. Since we can have multiple accumulators for each key, when this happens they are merged with `mergeCombiners`.

We can disable map side aggregation in `combineByKey` if we know that our data won't benefit from it. For example `groupByKey` disables map side aggregation as the aggregation function (appending to a list) does not save any data. If we want to disable map side combines we need to specify the partitioner, and for now you can just use the partitioner on the source rdd at `rdd.partitioner`.

Since `combineByKey` has a lot of different parameters it is a great candidate for an explanatory example. To better illustrate how `combineByKey` works we will look at computing the average value for each key.

Example 4-12. Python per-key average using combineByKey

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)),
                             (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.collectAsMap()
```

Example 4-13. Scala per-key average using combineByKey

```
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)
  ).map{ case (key, value) => (key, value._1 / value._2.toFloat) }
  result.collectAsMap().map(println(_))
```

Example 4-14. Java per-key average using combineByKey

```
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {
          total_ = total;
          num_ = num;
  }
  public int total_;
  public int num_;
  public float avg() {
```

```
        return total_ / (float) num_;
    }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, AvgCount>() {
    @Override
    public AvgCount call(Integer x) {
      return new AvgCount(x, 1);
    }
};
Function2<AvgCount, Integer, AvgCount> addAndCount =
    new Function2<AvgCount, Integer, AvgCount>() {
    @Override
    public AvgCount call(AvgCount a, Integer x) {
      a.total_ += x;
      a.num_ += 1;
      return a;
    }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
    new Function2<AvgCount, AvgCount, AvgCount>() {
    @Override
    public AvgCount call(AvgCount a, AvgCount b) {
      a.total_ += b.total_;
      a.num_ += b.num_;
      return a;
    }
};
AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
    nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
    System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}
```

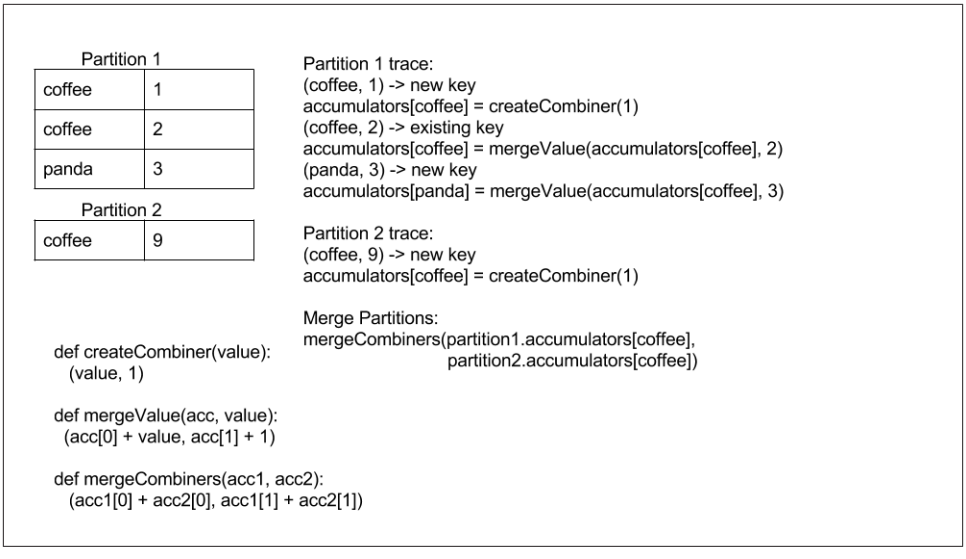

Figure 4-3. Combine by key sample data flow

There are many options for combining our data together by key. Most of them are implemented on top of `combineByKey` but provide a simpler interface. Using one of the specialized per-key combiners in Spark can be much faster than the naive approach of grouping our data and then reducing the data.

Tuning the Level of Parallelism

So far we have talked about how all of our transformations are distributed, but we have not really looked at how Spark decides how to split up the work. Every RDD has a fixed number of *partitions* which determine the degree of parallelism to use when executing operations on the RDD.

When performing aggregations or grouping operations, we can ask Spark to use a specific number of partitions. Spark will always try to infer a sensible default value based on the size of your cluster, but in some cases users will want to tune the level of parallelism for better performance.

Most of the operators discussed in this chapter accept a second parameter giving the number of partitions to use when creating the grouped or aggregated RDD:

Example 4-15. Python reduceByKey with custom parallelism

```
data = [("a", 3), ("b", 4), ("a", 1)]
sc.parallelize(data).reduceByKey(lambda x, y: x + y)      # Default parallelism
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)  # Custom parallelism
```

Example 4-16. Scala reduceByKey with custom parallelism

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey(_ + _)        // Default parallelism
sc.parallelize(data).reduceByKey(_ + _, 10)    // Custom parallelism
```

Sometimes, we want to change the partitioning of an RDD outside of the context of grouping and aggregation operations. For those cases, Spark provides the `repartition` function. Keep in mind that repartitioning your data is a fairly expensive operation. Spark also has an optimized version of `repartition` called `coalesce` that allows avoiding data movement, but only if you are decreasing the number of RDD partitions. To know whether you can safely call `coalesce` you can check the size of the RDD using `rdd.partitions.size()` in Java/Scala and `rdd.getNumPartitions()` in Python and make sure that you are coalescing it to fewer partitions that it currently has.

Grouping Data

With keyed data a common use case is grouping our data together by key, say joining all of a customers orders together.

If our data is already keyed in the way which we are interested `groupByKey` will group our data together using the key in our RDD. On an RDD consisting of keys of type K and values of type V we get back an RDD of type `[K, Iterable[V]]`.

`groupBy` works on unpaired data or data where we want to use a different condition besides equality on the current key. `groupBy` takes a function which it applies to every element in the source RDD and uses the result to determine the key.

If you find your self writing code where you `groupByKey` and then do a `reduce` or `fold` on the values, you can probably achieve the same result more efficiently by using one of the per-key combiners. Rather than reducing the RDD down to an in memory value, the data is reduced per-key and we get back an RDD with the reduced values corresponding to each key. e.g. `rdd.reduceByKey(func)` produces the same RDD as `rdd.groupByKey().mapValues(value => value.reduce(func))` but is more efficient as it avoids the step of creating a list of values for each key.

In addition to grouping together data from a single RDD, we can group together data sharing the same key from multiple RDDs using a function called `cogroup`. `cogroup` over two RDDs sharing the same key type K with the respective value types V and W gives us back `RDD[(K, Tuple(Iterable[V], Iterable[W]))]`. If one of the RDDs doesn't have elements for a given key that is present in the other RDD the corresponding

Iterable is simply empty. `cogroup` gives us the power to group together data from multiple RDDs.

The basic transformation on which the joins we discuss in the next section are implemented is `cogroup`. `cogroup` returns a Pair RDD with an entry for every key found in any of the RDDs it is called on along with a tuple of iterators with each iterator containing all of the elements in the corresponding RDD for that key.

`cogroup` can be used for much more than just implementing joins. We can also use it to implement intersect by key. Additionally, `cogroup` can work on three RDDs at once.

Joins

Some of the most useful operations we get with keyed data comes from using it together with other keyed data. Joining data together is probably one of the most common operations on a Pair RDD, and we have a full range of options including right and left outer joins, cross joins, inner joins.

The simple `join` operator is an inner join. Only keys which are present in both Pair RDDs are output. When there are multiple values for the same key in one of the inputs the resulting Pair RDD will also have multiple entries for the same key, with the values being the Cartesian product of the values for that key in each of the input RDDs. A simple way to understand this is by looking at an example of a join.

Example 4-17. Scala shell inner join example

```
storeAddress = {
  (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ness Ave"),
  (Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seattle")}

storeRating = {
  (Store("Ritual"), 4.9), (Store("Philz"), 4.8))}

storeAddress.join(storeRating) == {
  (Store("Ritual"), ("1026 Valencia St", 4.9)),
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),
  (Store("Philz"), ("3101 24th St", 4.8))}
```

Sometimes we don't need the key to be present in both RDDs to want it in our result. For example if we were joining customer information with recommendations we might not want to drop customers if there were not any recommendations yet. `leftOuterJoin(other)` and `rightOuterJoin(other)` both join Pair RDDs together by key where one of the Pair RDDs can be missing the key.

With `leftOuterJoin` the resulting Pair RDD has entries for each key in the source RDD. The value associated with each key in the result is a tuple of the value from the source RDD and an `Option` (or `Optional` in Java) for the value from the other Pair RDD. In Python if an value isn't present `None` is used and if the value is present the regular value, without any wrapper, is used. Like with `join` we can have multiple entries for each key and when this occurs we get the cartesian product between the two list of values.

`Optional` is part of Google's Guava library and is similar to nullable. We can check `isPresent()` to see if its set, and `get()` will return the contained instance provided it has data present.

`rightOuterJoin` is almost identical to `leftOuterJoin` except the key must be present in the other RDD and the tuple has an option for the source rather than other RDD.

We can look at our example from earlier and do a `leftOuterJoin` and a `rightOuterJoin` between the two Pair RDDs we used to illustrate `join`.

Example 4-18. Scala shell leftOuterJoin / rightOuterJoin examples

```
storeAddress.leftOuterJoin(storeRating) =
{(Store("Ritual"),("1026 Valencia St",Some(4.9))),
  (Store("Starbucks"),("Seattle",None)),
  (Store("Philz"),("748 Van Ness Ave",Some(4.8))),
  (Store("Philz"),("3101 24th St",Some(4.8)))}

storeAddress.rightOuterJoin(storeRating) =
{(Store("Ritual"),(Some("1026 Valencia St"),4.9)),
  (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),
  (Store("Philz"), (Some("3101 24th St"),4.8))}
```

Sorting Data

Having sorted data is quite useful in many cases, especially when producing downstream output. We can sort an RDD with key value pairs provided that there is an ordering defined on the key. Once we have sorted our data any subsequent call on the sorted data to `collect` or `save` will result in ordered data.

Since we often want our RDDs in the reverse order, the sortByKey function takes a parameter called `ascending` indicating if we want it in ascending order (defaults to `true`). Sometimes we want a different sort order entirely, and to support this we can provide our own comparison function here we will sort our RDD by converting the integers to strings and using the string comparison functions.

Example 4-19. Custom sort order in Python sorting integers as if strings

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda x: str(x))
```

Example 4-20. Custom sort order in Scala sorting integers as if strings

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
  override def compare(a: Int, b: Int) = a.toString.compare(b.toString)
}
rdd.sortByKey()
```

Example 4-21. Custom sort order in Java sorting integers as if strings

```
class IntegerComparator implements Comparator<Integer> {
   public int compare(Integer a, Integer b) {
     return String.valueOf(a).compareTo(String.valueOf(b))
  }
}
rdd.sortByKey(comp)
```

Actions Available on Pair RDDs

Like with the transformations, all of the traditional actions available on the base RDD are also available on Pair RDDs. Some additional actions are available on Pair RDDs which take advantage of the key-value nature of the data.

Table 4-3. Actions on Pair RDDs (example ({(1, 2), (3, 4), (3, 6)}))

countByKey()	Count the number of elements for each key	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Collect the result as a map to provide easy lookup	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Return all values associated with the provided key	rdd.lookup(3)	[4, 6]

There are also multiple other actions on Pair RDDs that save the RDD, which we will examine in the next chapter.

Data Partitioning

The final Spark feature we will discuss in this chapter is how to control datasets' partitioning across nodes. In a distributed program, communication is very expensive, so laying out data to minimize network traffic can greatly improve performance. Much like how a single-node program needs to choose the right data structure for a collection of records, Spark programs can choose to control their RDDs' partitioning to reduce communication. Partitioning will not be helpful in all applications — for example, if a given RDD is only scanned once, there is no point in partitioning it in advance. It is only useful when a dataset is reused *multiple times* in key-oriented operations such as joins. We will give some examples below.

Spark's partitioning is available on all RDDs of key-value pairs, and causes the system to group together elements based on a function of each key. Although Spark does not give explicit control of which worker node each key goes to (partly because the system is designed to work even if specific nodes fail), it lets the program ensure that a *set* of keys will appear together on *some* node. For example, one might choose to hash-partition an RDD into 100 partitions so that keys that have the same hash value modulo 100 appear on the same node. Or one might range-partition the RDD into sorted ranges of keys so that elements with keys in the same range appear on the same node.

As a simple example, consider an application that keeps a large table of user information in memory — say, an RDD of `(UserID, UserInfo)` pairs where `UserInfo` contains a list of topics the user is subscribed to. The application periodically combines this table with a smaller file representing events that happened in the past five minutes — say, a table of `(UserID, LinkInfo)` pairs for users who have clicked a link on a website in those five minutes. For example, we may wish to count how many users visited a link that was *not* to one of their subscribed topics. We can perform this combining with Spark's `join` operation, which can be used to group the `UserInfo` and `LinkInfo` pairs for each `UserID` by key. Our application would look like this:

```
// Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.
// This distributes elements of userData by the HDFS block where they are found,
// and doesn't provide Spark with any way of knowing in which partition a
// particular UserID is located.
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

// Function called periodically to process a log file of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events)// RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

This code will run fine as is, but it will be inefficient. This is because the `join` operation, called each time that `processNewLogs` is invoked, does not know anything about how the keys are partitioned in the datasets. By default, this operation will hash all the keys of *both* datasets, sending elements with the same key hash across the network to the same machine, and then join on that machine the elements with the same key (to come). Because we expect the `userData` table to be much larger than the small log of `events` seen every five minutes, this wastes a lot of work: the `userData` table is hashed and shuffled across the network on every call, even though it doesn't change.

Fixing this is simple: just use the `partitionBy` transformation on `userData` to hash-partition it at the start of the program. We do this by passing a `spark.HashPartitioner` object to `partitionBy`:

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100))   // Create 100 partitions
                 .persist()
```

The `processNewLogs` method can remain unchanged — the `events` RDD is local to `processNewLogs`, and is only used once within this method, so there is no advantage in specifying a partitioner for `events`. Because we called `partitionBy` when building `userData`, Spark will now know that it is hash-partitioned, and calls to `join` on it will take advantage of this information. In particular, when we call `userData.join(events)`, Spark will only shuffle the `events` RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData` (to come). The result is that a lot less data is communicated over the network, and the program runs significantly faster.

Note that `partitionBy` is a transformation, so it always returns a *new* RDD — it does not change the original RDD in-place. RDDs can never be modified once created. Therefore it is important to persist and save as `userData` the result of `partitionBy`, not the original `sequenceFile`. Also, the 100 passed to `partitionBy` represents the number of partitions, which will control how many parallel tasks perform further operations on the RDD (e.g., joins); in general, make this at least as large as the number of cores in your cluster.

Failure to `persist` an RDD after it has been transformed with `partitionBy` will cause subsequent uses of the RDD to repeat the partitioning of the data. Without persistence, use of the partitioned RDD will cause re-evaluation of the RDDs complete lineage. That would negate the advantage of `partitionBy`, resulting in repeated partitioning and shuffling of data across the network, similar to what occurs without any specified partitioner.

When using a `HashPartitioner`, specify a number of hash buckets at least as large as the number of cores in your cluster in order to achieve appropriate parallelism.

In fact, many other Spark operations automatically result in an RDD with known partitioning information; and many operations other than `join` will take advantage of this information. For example, `sortByKey` and `groupByKey` will result in range-partitioned

and hash-partitioned RDDs, respectively. On the other hand, operations like `map` cause the new RDD to *forget* the parent's partitioning information, because such operations could theoretically modify the key of each record. The next few sections describe how to determine how an RDD is partitioned, and exactly how partitioning affects the various Spark operations.

Partitioning in Java and Python

Spark's Java and Python APIs benefit from partitioning the same way as the Scala API. However, in Python, you cannot pass a `HashPartitioner` object to `partitionBy`; instead, you just pass the number of partitions desired (e.g., `rdd.partitionBy(100)`).

Determining an RDDs Partitioner

In Scala and Java, you can determine how an RDD is partitioned using its `partitioner` property (or `partitioner()` method in Java).[1] This returns a `scala.Option` object, which is a Scala class for a container that may or may not contain one item. (It is considered good practice in Scala to use `Option` for fields that may not be present, instead of setting a field to `null` if a value is not present, running the risk of a null-pointer exception if the program subsequently tries to use the `null` as if it were an actual, present value.) You can call `isDefined()` on the `Option` to check whether it has a value, and `get()` to get this value. If present, the value will be a `spark.Partitioner` object. This is essentially a function telling the RDD which partition each key goes into — more about this later.

The `partitioner` property is a great way to test in the Spark shell how different Spark operations affect partitioning, and to check that the operations you want to do in your program will yield the right result. For example:

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> pairs.partitioner
res0: Option[spark.Partitioner] = None

scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <console>:14

scala> partitioned.partitioner
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

In this short session, we created an RDD of `(Int, Int)` pairs, which initially have no partitioning information (an `Option` with value `None`). We then created a second RDD

1. The Python API does not yet offer a way to query partitioners, though it still uses them internally.

by hash-partitioning the first. If we actually wanted to use `partitioned` in further operations, then we should have appended `.cache()` to the third line of input, in which `partitioned` is defined. This is for the same reason that we needed `cache` for `userData` in the previous example: without `cache` or `persist`, subsequent RDD actions will evaluate the entire lineage of `partitioned`, which will cause `pairs` to be hash-partitioned over and over.

Operations that Benefit from Partitioning

Many of Spark's operations involve shuffling data by key across the network. All of these will benefit from partitioning. As of Spark 1.0, the operations that benefit from partitioning are: `cogroup`, `groupWith`, `join`, `leftOuterJoin`, `rightOuterJoin`, `groupByKey`, `reduceByKey`, `combineByKey`, and `lookup`.

For operations that act on a single RDD, such as `reduceByKey`, running on a pre-partitioned RDD will cause all the values for each key to be computed *locally* on a single machine, requiring only the final, locally-reduced value to be sent from each worker node back to the master. For binary operations, such as `cogroup` and `join`, pre-partitioning will cause at least one of the RDDs (the one with the known partitioner) to not be shuffled. If both RDDs have the *same* partitioner, and if they are cached on the same machines (e.g. one was created using `mapValues` on the other, which preserves keys and partitioning) or if one of them has not yet been computed, then no shuffling across the network will occur.

Operations that Affect Partitioning

Spark knows internally how each of its operations affects partitioning, and automatically sets the `partitioner` on RDDs created by operations that partition the data. For example, suppose you called `join` to join two RDDs; because the elements with the same key have been hashed to the same machine, Spark knows that the result is hash-partitioned, and operations like `reduceByKey` on the join result are going to be significantly faster.

The flip-side, however, is that for transformations that *cannot* be guaranteed to produce a known partitioning, the output RDD will not have a `partitioner` set. For example, if you call `map` on a hash-partitioned RDD of key-value pairs, the function passed to `map` can in theory change the key of each element, so the result will not have a `partitioner`. Spark does not analyze your functions to check whether they retain the key. Instead, it provides two other operations, `mapValues` and `flatMapValues`, which guarantee that each tuple's key remains the same.

All that said, here are all the operations that result in a partitioner being set on the output RDD: `cogroup`, `groupWith`, `join`, `leftOuterJoin`, `rightOuterJoin`, `groupByKey`, `reduceByKey`, `combineByKey`, `partitionBy`, `sort`, `mapValues` (if the parent RDD has a

partitioner), `flatMapValues` (if parent has a partitioner), and `filter` (if parent has a partitioner). All *other* operations will produce a result with no partitioner.

That there is a partitioner does not answer the question of *what* the output partitioner is for binary operations such as `join`. By default, it is a hash partitioner, with the number of partitions set to the level of parallelism of the operation. However, if one of the parents has a `partitioner` object, it will be that partitioner; and if both parents have a `partitioner` set, it will be the partitioner of the first parent (the one that `join` was called on, for example).

Example: PageRank

As an example of a more involved algorithm that can benefit from RDD partitioning, we consider PageRank. The PageRank algorithm, named after Google's Larry Page, aims to assign a measure of importance (a "rank") to each document in a set based on how many documents have links to it. It can be used to rank web pages, of course, but also scientific articles, or influential users in a social network (by treating each user as a "document" and each friend relationship as a "link").

PageRank is an iterative algorithm that performs many joins, so it is a good use case for RDD partitioning. The algorithm maintains two datasets: one of `(pageID, linkList)` elements containing the list of neighbors of each page, and one of `(pageID, rank)` elements containing the current rank for each page. It proceeds as follows:

1. Initialize each page's rank to 1.0
2. On each iteration, have page `p` send a contribution of `rank(p) / numNeighbors(p)` to its neighbors (the pages it has links to).
3. Set each page's rank to `0.15 + 0.85 * contributionsReceived`.

The last two steps repeat for several iterations, during which the algorithm will converge to the correct PageRank value for each page. As a simple solution, it's typically enough to run about ten iterations and declare the resulting ranks to be the PageRank values.

Here is the code to implement PageRank in Spark:

```
val sc = new SparkContext(...)

// Assume that our neighbor list was saved as a Spark objectFile
val links = sc.objectFile[(String, Seq[String])]("links")
              .partitionBy(new HashPartitioner(100))
              .persist()

// Initialize each page's rank to 1.0; since we use mapValues, the resulting RDD
// will have the same partitioner as links
var ranks = links.mapValues(_ => 1.0)
```

```
// Run 10 iterations of PageRank
for (i <- 0 until 10) {
  val contributions = links.join(ranks).flatMap {
    case (pageId, (links, rank)) =>
      links.map(dest => (dest, rank / links.size))
  }
  ranks = contributions.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
}

// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

That's it! The algorithm starts with a `ranks` RDD initialized at 1.0 for each element, and keeps updating the `ranks` variable on each iteration. The body of PageRank is pretty simple to express in Spark: it first does a `join` between the current `ranks` RDD and the static `links` one, in order to obtain the link list and rank for each page ID together, then uses this in a `flatMap` to create "contribution" values to send to each of the page's neighbors. We then add up these values by page ID (i.e. by the page receiving the contribution) and set that page's rank to `0.15 + 0.85 * contributionsReceived`.

Although the code itself is simple, the example does several things to ensure that the RDDs are partitioned in an efficient way, and to minimize communication:

1. Notice that the `links` RDD is joined against `ranks` on each iteration. Since `links` is a static dataset, we partition it at the start with `partitionBy`, so that it does not need to be shuffled across the network. In practice, the `links` RDD is also likely to be much larger in terms of bytes than `ranks`, since it contains a list of neighbors for each page ID instead of just a `Double`, so this optimization saves considerable network traffic over a simple implementation of PageRank (e.g. in plain MapReduce).
2. For the same reason, we call `persist` on `links` to keep it in RAM across iterations.
3. When we first create `ranks`, we use `mapValues` instead of `map` to preserve the partitioning of the parent RDD (`links`), so that our first join against it is very cheap.
4. In the loop body, we follow our `reduceByKey` with `mapValues`; because the result of `reduceByKey` is already hash-partitioned, this will make it more efficient to join the mapped result against `links` on the next iteration.

Finally, note also that the extra syntax from using partitioning is small, and `mapValues` in particular is more concise in the places it's used here than a `map`.

To maximize the potential for partitioning-related optimizations, you should use `mapValues` or `flatMapValues` whenever you are not changing an element's key.

Custom Partitioners

While Spark's `HashPartitioner` and `RangePartitioner` are well-suited to many use cases, Spark also allows you to tune how an RDD is partitioned by providing a custom `Partitioner` object. This can be used to further reduce communication by taking advantage of domain-specific knowledge.

For example, suppose we wanted to run the PageRank algorithm in the previous section on a set of web pages. Here each page's ID (the key in our RDD) will be its URL. Using a simple hash function to do the partitioning, pages with similar URLs (e.g., `http://www.cnn.com/WORLD` and `http://www.cnn.com/US`) might be hashed to completely different nodes. However, we know that web pages within the same domain tend to link to each other a lot. Because PageRank needs to send a message from each page to each of its neighbors on each iteration, it helps to group these pages into the same partition. We can do this with a custom `Partitioner` that looks at just the domain name instead of the whole URL.

To implement a custom partitioner, you need to subclass the `spark.Partitioner` class and implement three methods:

- `numPartitions: Int`, which returns the number of partitions you will create
- `getPartition(key: Any): Int`, which returns the partition ID (0 to `numPartitions-1`) for a given key
- `equals`, the standard Java equality method. This is important to implement because Spark will need to test your `Partitioner` object against other instances of itself when it decides whether two of your RDDs are partitioned in the same way!

One gotcha is that, if you rely on Java's `hashCode` method in your algorithm, it can return negative numbers. You need to be careful that `getPartition` always returns a nonnegative result.

For example, here is how we would write the domain-name based partitioner sketched above, which hashes only the domain name of each URL:

```
class DomainNamePartitioner(numParts: Int) extends Partitioner {
  override def numPartitions: Int = numParts

  override def getPartition(key: Any): Int = {
    val domain = new java.net.URL(key.toString).getHost()
    val code = (domain.hashCode % numPartitions)
    if (code < 0) {
      code + numPartitions  // Make it non-negative
    } else {
      code
    }
  }
```

```
    // Java equals method to let Spark compare our Partitioner objects
    override def equals(other: Any): Boolean = other match {
      case dnp: DomainNamePartitioner =>
        dnp.numPartitions == numPartitions
      case _ =>
        false
    }
  }
```

Note that in the `equals` method, we used Scala's pattern matching operator (`match`) to test whether `other` is a `DomainNamePartitioner`, and cast it if so; this is the same as using `instanceof` in Java.

Using a custom `Partitioner` is easy: just pass it to the `partitionBy` method. Many of the shuffle-based methods in Spark, such as `join` and `groupByKey`, can also take an optional `Partitioner` object to control the partitioning of the output.

Creating a custom `Partitioner` in Java is very similar to Scala: just extend the `spark.Partitioner` class and implement the required methods.

In Python, you do not extend a `Partitioner` class, but instead pass a hash function as an additional argument to `RDD.partitionBy()`. For example:

```
import urlparse

def hash_domain(url):
  return hash(urlparse.urlparse(url).netloc)

rdd.partitionBy(20, hash_domain)  # Create 20 partitions
```

Note that the hash function you pass will be compared *by identity* to that of other RDDs. If you want to partition multiple RDDs with the same partitioner, pass the same function object (e.g., a global function) instead of creating a new `lambda` for each one!

Conclusion

In this chapter, we have seen how to work with keyed data using the specialized functions available in Spark. The techniques from the previous chapter on Programming with RDDs also still work on our Pair RDDs. In the next chapter, we will look at how to load and save data.

CHAPTER 5
Loading and Saving Your Data

Both engineers and data scientists will find parts of this chapter useful. Engineers may wish to explore more output formats to see if there is something well suited to their intended down stream consumer available and should consider looking online for different Hadoop formats. Data Scientists can likely focus on the input format that their data is already in. Spark supports reading from classes that implement Hadoop's InputFormat and writing to Hadoop's OutputFormat interfaces.

Motivation

We've looked at a number of operations we can perform on our data once we have it distributed in Spark. So far our examples have loaded and saved all of their data from a native collection and regular files, but odds are that your data doesn't fit on a single machine, so its time to explore our options for loading and saving.

It is important for the machines in our cluster to be able to access the files. There are a variety of different distributed file systems we can use which we cover in "File Systems" on page 84, including S3 and HDFS.

Different compression options can decrease the amount of space and network overhead required but can introduce restrictions on how we read the data. These compression options are covered in "Compression" on page 85.

Often when we need data for our code we don't get to choose the format. Sometimes we are lucky enough to be able to choose from multiple formats or work with our upstream data providers. We will start with some common file formats which we can write to a number of different file systems. In addition to standard file formats, we will also examine using different database systems for IO. Spark works with all the formats implementing Hadoop's InputFormat and OutputFormat interfaces, see "Hadoop Input and Output Formats" on page 81.

In Spark 1.0, the standard data APIs for Python only support text data. However, you can work around this by using Spark SQL, which allows loading data from a number of formats supported by Hive. We will cover Spark SQL in a later chapter, but briefly illustrate how to load data from Hive in this chapter too.

In addition to these file formats, Spark can also work directly with a number of database and search systems. We will look at Mongo, Hive, Cassandra, and Elasticsearch, but many more are supported through the standard Hadoop connectors with Spark. This is convenient if your data happens to already be in one of these systems as nightly dumps of data can be difficult to maintain. When about to run a new Spark job against an online system, read or write, you should verify you have sufficient capacity for a potentially very high volume of queries and updates footenote:[In general think if your data sources or data sink can handle the number of readers/writers as the number of partitions you have.].

Formats

Spark makes it very simple to load and save data in a large number of formats. Formats range from unstructured, like text, to semi-structured, like JSON, and to structured like Sequence Files. The input formats that Spark wraps all transparently handle compressed formats based on the file extension.

Table 5-1. Common Supported File Formats

Format name	Splittable	Structured	comments
text files	yes	no	Plain old text files. Splittable provided records are one per line.
JSON	yes	semi	Common text based format, are semi-structured, splittable if one record per line.
CSV	yes	yes	Very common text based format, often used with spreadsheet applications.
Sequence files	yes	yes	A common Hadoop file format used for key-value data.
Protocol Buffers	yes	yes	A fast space-efficient multi-language format.
Object Files	yes	yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

In addition to the output mechanism supported directly in Spark, we can use both Hadoop's new and old file APIs for keyed (or paired) data. We can only use these with key-value data, because the Hadoop interfaces requires key-value data, even though some formats ignore the key. In cases where the format ignores the key it is common to use a dummy key (such as null).

Text Files

Text files are very simple to load from and save to with Spark. When we load a single text file as an RDD each input line becomes an element in the RDD. We can also load multiple whole text files at the same time into a Pair RDD with the key being the name and the value being the contents of each file.

Loading Text Files

Loading a single text file is as simple as calling the `textFile` function on our spark context with the path to the file. If we want to control the number of partitions we can also specify minPartitions.

Example 5-1. Python load text file example

```
input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

Example 5-2. Scala load text file example

```
val input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

Example 5-3. Java load text file example

```
JavaRDD<String> input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

Multi-part inputs in the form of a directory containing all of the parts can be handled in two ways. We can just use the same `textFile` method and pass it a directory and it will load all of the parts into our RDD. Sometimes its important to know which file which piece of input came from (such as time data with the key in the file) or we need to process an entire file at a time. If our files are small enough then we can use the `wholeTextFiles` method and get back a Pair RDD where the key is the name of the input file.

`wholeTextFiles` can be very useful when each file represents a certain time periods data. If we had files representing sales data from different periods we could easily compute the average for each period.

Example 5-4. Scala average value per file

```
val input = sc.wholeTextFiles("file://home/holden/happypanda")
val result = input.mapValues{y =>
  val nums = y.split(" ").map(x => x.toDouble)
  nums.sum / nums.size.toDouble
}
```

Spark supports reading all the files in a given directory and doing wild card expansion on the input (e.g. `part-*.txt`). This is useful since large data sets are often spread across multiple files, especially if other files (like success markers) may be in the same directory.

Saving Text Files

Outputting text files is also quite simple. The method `saveAsTextFile` takes a path and will output the contents of the RDD to that file. The path is treated as a directory and Spark will output multiple files underneath that directory. This allows Spark to write the output from multiple nodes. With this method we don't get to control which files end up with which segments of our data but there are other output formats which do allow this.

Example 5-5. Scala save as text file example

```
result.saveAsTextFile(outputFile)
```

Loading and saving text files is implemented through wrappers around Hadoop's file APIs. If you want further examples of how to work with the Hadoop file APIs you can see how this is implemented in SparkContext and RDD respectively.

JSON

JSON is a popular semi-structured data format. The simplest way to load JSON data is by loading the data as a text file and then mapping over the values with a JSON parser. Like wise, we can use our preferred JSON serialization library to write out the values to strings which we can then write out. In Java and Scala we can also work with JSON data using a custom Hadoop format. The (to come) also shows how to load JSON data with Spark SQL.

Loading JSON

Loading the data as a text file and then parsing the JSON data is an approach that we can use in all of the supported languages. This works assuming that you have one JSON record per-row, if you have multi-line JSON files you will instead have to load the whole file and then parse each file. This is the same technique we use with "Loading CSV" on page 75 except we use JSON parser. If constructing a JSON parser is expensive in your language, you can use `mapPartitions` to re-use the parser, see "Working on a Per-Partition Basis" on page 100 for details.

There are a wide variety of JSON libraries available for the three languages we are looking at, for simplicity's sake we are only considering one library per language. In Java and Scala we will use Jackson, and in python we use the built in library. These libraries have been chosen as they perform reasonable well and are also relatively simple, if you spend

a lot of time in the parsing stage you may wish to look at other JSON libraries for Scala or for Java.

Example 5-6. Python load unstructured JSON example

```
data = input.map(lambda x: json.loads(x))
```

If your JSON data happens to follow a predictable schema (lucky you!), we can parse it into a more structured format. This is often where we will find invalid records, so we may wish to skip them.

Example 5-7. Scala load JSON example

```
import com.fasterxml.jackson.module.scala.DefaultScalaModule
import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.DeserializationFeature
...
case class Person(name: String, lovesPandas: Boolean) // Note: must be a top level class
...
// Parse it into a specific case class. We use flatMap to handle errors
// by returning an empty list (None) if we encounter an issue and a
// list with one element if everything is ok (Some(_)).
val result = input.flatMap(record => {
  try {
    Some(mapper.readValue(record, classOf[Person]))
  } catch {
    case e: Exception => None
  }})
```

Example 5-8. Java load JSON example

```
public static class ParseJson implements FlatMapFunction<Iterator<String>, Person> {
  public Iterable<Person> call(Iterator<String> lines) throws Exception {
    ArrayList<Person> people = new ArrayList<Person>();
    ObjectMapper mapper = new ObjectMapper();
    while (lines.hasNext()) {
      String line = lines.next();
      try {
        people.add(mapper.readValue(line, Person.class));
      } catch (Exception e) {
        // skip records on failure
      }
    }
    return people;
  }
}
```

Handling incorrectly formatted records can be a big problem, especially with semi-structured data like JSON. With small data sets it can be acceptable to stop the world (i.e. fail the program) on malformed input, but often with large data sets malformed input is simply a part of life. If you do choose to skip incorrectly formatted (or attempt to recover) incorrectly formatted data you may wish to look at using accumulators to keep track of the number of errors.

Saving JSON

Writing out JSON files is much simpler compared to loading it as we don't have to worry about incorrectly formatted data and we know the type of the data that we are writing out. We can use the same libraries we used to convert our RDD of strings into parsed JSON data and instead take our RDD of structured data and convert it into an RDD of strings which we can then write out using Spark's text file API.

Lets say we were running a promotion for people that love pandas, so we can take our input from the first step and filter it for the people that love pandas.

Example 5-9. Python save JSON example

```
(data.filter(lambda x: x['lovesPandas']).map(lambda x: json.dumps(x))
  .saveAsTextFile(outputFile))
```

Example 5-10. Scala save JSON example

```
result.filter(_.lovesPandas).map(mapper.writeValueAsString(_))
  .saveAsTextFile(outputFile)
```

Example 5-11. Java save JSON example

```
public static class WriteJson implements
  FlatMapFunction<Iterator<Person>, String> {
  public Iterable<String> call(Iterator<Person> people) throws Exception {
    ArrayList<String> text = new ArrayList<String>();
    ObjectMapper mapper = new ObjectMapper();
    while (people.hasNext()) {
      Person person = people.next();
      text.add(mapper.writeValueAsString(person));
    }
    return text;
  }
}

JavaRDD<Person> result = input.mapPartitions(new ParseJson()).filter(
  new LikesPandas());
JavaRDD<String> formatted = result.mapPartitions(new WriteJson());
formatted.saveAsTextFile(outfile);
```

We can easily load and save JSON data with Spark by using the existing mechanism for working with text and adding JSON libraries.

CSV (Comma Separated Values) / TSV (Tab Separated Values)

CSV files are supposed to contain a fixed number of fields per-line, and the fields are most commonly separated by comma or tab. Records are often stored one per line, but this is not always the case as records can sometimes span lines. CSV/TSV files can sometimes be inconsistent, most frequently in respect to handling newlines, escaping, non-ASCII characters, non-integer numbers. CSVs cannot handle nested field types natively, so we have to unpack and pack to specific fields manually.

Unlike with JSON fields each record doesn't have field names associated with them; instead we get back row numbers. It is common practice in single CSV files to have the first rows column values be the names of each field.

Loading CSV

Loading CSV/TSV data is similar to loading JSON data in that we can first load it as text and then process it. The lack of standardization of format leads to different versions of the same library sometimes handling input in different ways.

Like with JSON there are many different CSV libraries and we will only use one for each language. In both Scala and Java we use opencsv. Once again in Python we use the included csv library.

As with JSON there is a Hadoop InputFormat, CSVInputFormat, that we can use to load CSV data in Scala and Java (although it does not support records containing newlines).

If your CSV data happens to not contain newlines in any of the fields, you can load your data with `textFile` and parse it.

Example 5-12. Python load CSV example

```
import csv
import StringIO
...
def loadRecord(line):
    """Parse a CSV line"""
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input, fieldnames=["name", "favouriteAnimal"])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

Example 5-13. Java load CSV example

```
import au.com.bytecode.opencsv.CSVReader;
import java.io.StringReader;
...
public static class ParseLine implements Function<String, String[]> {
  public String[] call(String line) throws Exception {
    CSVReader reader = new CSVReader(new StringReader(line));
    return reader.readNext();
  }
}
JavaRDD<String> csvFile1 = sc.textFile(csv1);
JavaPairRDD<String[]> csvData = csvFile1.map(new ParseLine());
```

Example 5-14. Scala load CSV example

```
import java.io.StringReader
import au.com.bytecode.opencsv.CSVReader
...
val input = sc.textFile(inputFile)
val result = input.map{ line =>
  val reader = new CSVReader(new StringReader(line));
  reader.readNext();
}
```

If there are embedded newlines in fields we will need to load each file in full and parse the entire segment. This is unfortunate because if each file is large this can easily introduce bottlenecks in loading and parsing. The different text file loading methods are described "Loading Text Files" on page 71.

Example 5-15. Python load CSV example

```
def loadRecords(fileNameContents):
    """Load all the records in a given file"""
    input = StringIO.StringIO(fileNameContents[1])
    reader = csv.DictReader(input, fieldnames=["name", "favoriteAnimal"])
    return reader
fullFileData = sc.wholeTextFiles(inputFile).flatMap(loadRecords)
```

Example 5-16. Java load CSV example

```
public static class ParseLine implements FlatMapFunction<Tuple2<String, String>, String[]> {
  public Iterable<String[]> call(Tuple2<String, String> file) throws Exception {
    CSVReader reader = new CSVReader(new StringReader(file._2()));
    return reader.readAll();
  }
}
JavaPairRDD<String, String> csvData = sc.wholeTextFiles(csvInput);
JavaRDD<String[]> keyedRDD = csvData.flatMap(new ParseLine());
```

Example 5-17. Scala load CSV example

```
case class Person(name: String, favoriteAnimal: String)

val input = sc.wholeTextFiles(inputFile)
val result = input.flatMap{ case (_, txt) =>
  val reader = new CSVReader(new StringReader(txt));
  reader.readAll().map(x => Person(x(0), x(1)))
}
```

If there are only a few input files, and you need to use the wholeFile method, you may want to repartition your input to allow Spark to effectively parallelize your future operations.

Saving CSV

As with JSON data, writing out CSV/TSV data is quite simple and we can benefit from reusing the output encoding object. Since in CSV we don't output the field name with each record, to have a consistent output we need to create a mapping. One of the easy ways to do this is to just write a function which converts the fields to given positions in an array. In Python if we are outputting dictionaries the csv writer can do this for us based on the order we provide the `fieldnames` when constructing the writer.

The CSV libraries we are using output to files/writers so we can use StringWriter/StringIO to allow us to put the result in our RDD.

Example 5-18. Python write csv example

```
def writeRecords(records):
    """Write out CSV lines"""
    output = StringIO.StringIO()
    writer = csv.DictWriter(output, fieldnames=["name", "favoriteAnimal"])
    for record in records:
        writer.writerow(record)
    return [output.getvalue()]

pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFile)
```

Example 5-19. Scala write CSV example

```
pandaLovers.map(person => List(person.name, person.favoriteAnimal).toArray)
.mapPartitions{people =>
  val stringWriter = new StringWriter();
  val csvWriter = new CSVWriter(stringWriter);
  csvWriter.writeAll(people.toList)
  Iterator(stringWriter.toString)
}.saveAsTextFile(outFile)
```

As you may have noticed the above only works provided that we know all of the fields that we will be outputting. However, if some of the field names are determined at runtime from user input we need to take a different approach. The simplest approach is going over all of our data and extracting the distinct keys and then taking another pass for output.

Sequence Files

Sequence files are a popular Hadoop format comprised of flat files with key-value pairs and are supported in Spark's Java and Scala APIs. Sequence files have sync markers that allow Spark to seek to a point in the file and then resychronize with the record boundaries. This allows Spark to efficiently read Sequence files in from multiple nodes and in to many partitions. Sequence Files are a common input/output format for Hadoop MapReduce jobs as well so if you are working with an existing Hadoop system there is a good change your data will be available as a sequence file.

Sequence files consist of elements which implement Hadoop's Writable interface, as Hadoop uses a custom serialization framework. We have a conversion table of some common types and their corresponding Writable class. The standard rule of thumb is try adding the word Writable to the end of your class name and see if it is a known subclass of org.apache.hadoop.io.Writable. If you can't find a Writable for the data you are trying to write out (like for example a custom case class), you can go ahead and implement your own Writable class by overriding readFields and write from org.apache.hadoop.io.Writable.

Hadoop's RecordReader re-uses the same object for each record, so directly calling `cache`, on an RDD you read in like this can fail, instead add a simple `map` operation and cache the result of the `map`. Further more, many Hadoop Writable classes do not implement `java.io.Serializable` so for them to work in RDDs we need to convert them with a `map` anyways.

Table 5-2. Corresponding Hadoop Writable Types

Scala Type	Java Type	Hadoop Writable
Int	Integer	IntWritable or VIntWritable [a]
Long	Long	LongWritable or VLongWritable [a]
Float	Float	FloatWritable
Double	Double	DoubleWritable
Boolean	Boolean	BooleanWritable
Array[Byte]	Byte[]	BytesWritable
String	String	Text

Scala Type	Java Type	Hadoop Writable
Array[T]	T[]	ArrayWritable<TW> [b]
List[T]	List<T>	ArrayWritable<TW> [b]
Map[A, B]	Map<A, B>	MapWritable<AW, BW> [b]

[a] ints and longs are often stored as a fixed size. Storing the number 12 takes the same amount of space as storing the number 2**30. If you might have a large number of small numbers. Instead we can use variable sized types which will use less bits to store smaller numbers.

[b] The templated type must also be a writable type.

Loading Sequence Files

Spark has a specialized API for reading in sequence files. On the Spark Context we can call `sequenceFile(path, keyClass, valueClass, minPartitions)`. As mentioned earlier, Sequence Files work with Writable classes, so our `keyClass` and `valueClass` will both have to be the correct Writable class. Lets consider loading people and the number of pandas they have seen from a sequence file, in this case our `keyClass` would be `Text` and our `valueClass` would be `IntWritable` or `VIntWritable`, for simplicity lets work with `IntWritable`.

Example 5-20. Scala load sequence file example

```
val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable]).
  map{case (x, y) => (x.toString, y.get())}
```

Example 5-21. Java load sequence file example

```
public static class ConvertToNativeTypes implements
  PairFunction<Tuple2<Text, IntWritable>, String, Integer> {
  public Tuple2<String, Integer> call(Tuple2<Text, IntWritable> record) {
    return new Tuple2(record._1.toString(), record._2.get());
  }
}

JavaPairRDD<Text, IntWritable> input = sc.sequenceFile(fileName, Text.class,
  IntWritable.class);
JavaPairRDD<String, Integer> result = input.mapToPair(
  new ConvertToNativeTypes());
```

In Scala there is a convenience function which can automatically convert Writables to their corresponding Scala type. Instead of specifying the keyClass and valueClass we can call `sequenceFile[Key, Class](path, minPartitions)` and get back an RDD of native Scala types.

Saving Sequence Files

Writing the data out to a sequence file is fairly similar in Scala. First since sequence files are key-value pairs, we need a `PairRDD` with types that our sequence file can write out. Implicit conversions between Scala types and Hadoop Writables exist for many native types, so if you are writing out a native type you can just save your `PairRDD` by calling `saveAsSequenceFile(path)` and it will write out the data for us. If there isn't an automatic conversion from our key and value to Writable, or we want to use VarInts we can just map over the data and convert it before saving. Lets consider writing out the data that we loaded in the previous example (people and how many pandas they have seen).

Example 5-22. Scala save sequence file example

```
val data = sc.parallelize(List(("Panda", 3), ("Kay", 6), ("Snail", 2)))
data.saveAsSequenceFile(outputFile)
```

In Java saving a sequence file is slightly more involved, due to the lack of `saveAsSequenceFile` method on the `JavaPairRDD`. Instead we use Spark's ability to save to custom Hadoop formats and we will show how to save to a sequence file in java in the custom Hadoop formats subsection.

Object Files

Object files are a deceptively simple wrapper around sequence files which allows us to save our RDDs containing just values. Unlike with Sequence files, the values are written out using Java Serialization.

If you change your classes, e.g., to add and remove fields, old object files may no longer be readable. Object files use Java Serialization, which has some support for managing compatibility across class versions but requires programmer effort to do so.

Using Java Serialization for object files has a number of implications. Unlike with normal sequence files, the output will be different than Hadoop outputting the same objects. Unlike the other formats, object files are mostly intended to be used for Spark jobs communicating with other Spark Jobs. Java Serialization can also be quite slow.

Saving an object file is as simple as calling `saveAsObjectFile` on an RDD. Reading an object file back is also quite simple, the function `objectFile` on the SparkContext takes in a path and returns an RDD.

With all of these warnings about object files you might wonder why anyone would use them. The primary reason to use object files are they require almost no work to save almost arbitrary objects.

Hadoop Input and Output Formats

In addition to the formats Spark has wrappers for, we can also interact with other Hadoop supported formats. Spark supports both the old and new Hadoop file APIs providing a great amount of flexibility.

Loading with other Hadoop Input Formats

To read in a file using the new Hadoop API we need to tell spark a few things. The `newAPIHadoopFile` takes a path, and three classes. The first class is the "format" class, this is the class representing our input format. The next class is the class for our key, and the final class is the class of our value. If we need to specify additional Hadoop configuration properties we can also pass in a `conf` object.

One of the simplest Hadoop input formats is the `KeyValueTextInputFormat` which can be used for reading in key-value data from text files. Each line is processed individually with they key and value separated by a tab character. This format ships with Hadoop so we don't have to add any extra dependencies to our project to use it.

Example 5-23. Scala load KeyValueTextInputFormat

```
val input = sc.hadoopFile[Text, Text, KeyValueTextInputFormat](inputFile).map{
  case (x, y) => (x.toString, y.toString)
}
```

We looked at loading JSON data by loading the data as a text file and then parsing it, but we can also load JSON data using a custom Hadoop input format. This example requires setting up some extra bits for compression so feel free to skip it. Twitter's Elephant Bird package supports a large number of data formats, including JSON, Lucene, Protocol Buffer related formats, and so on. The package also works with both the new and old Hadoop file APIs. To illustrate how to work with the new style Hadoop APIs from Spark lets look at loading LZO compressed JSON data with `LzoJsonInputFormat`:

Example 5-24. Scala load LZO compressed JSON with Elephant Bird

```
val input = sc.newAPIHadoopFile(inputFile, classOf[LzoJsonInputFormat],
  classOf[LongWritable], classOf[MapWritable], conf)
// Each MapWritable in "input" represents a JSON object
```

LZO support requires installing the `hadoop-lzo` package and pointing Spark to its native libraries. If you install the Debian package, adding `--driver-library-path /usr/lib/hadoop/lib/native/ --driver-class-path /usr/lib/hadoop/lib/` to your `spark-submit` invocation should do the trick.

Reading a file using the old Hadoop API is pretty much the same from a usage point of view, except we provide an old style InputFormat class. Many of Spark's built in convenience functions (like `sequenceFile`) are implemented using the old style Hadoop API.

Saving with Hadoop Output Formats

We already examined sequence files to some extent, but in Java we don't have the same convience function for saving from a `PairRDD`. We will use this as a way to illustrate how to have using the old Hadoop format APIs as this is how Spark implements its helper function for PairRDDs in scala and we've already shown the new APIs with the JSON example.

Example 5-25. Java save sequence file example

```
public static class ConvertToWritableTypes implements
  PairFunction<Tuple2<String, Integer>, Text, IntWritable> {
  public Tuple2<Text, IntWritable> call(Tuple2<String, Integer> record) {
    return new Tuple2(new Text(record._1), new IntWritable(record._2));
  }
}

JavaPairRDD<String, Integer> rdd = sc.parallelizePairs(input);
JavaPairRDD<Text, IntWritable> result = rdd.mapToPair(new ConvertToWritableTypes());
result.saveAsHadoopFile(fileName, Text.class, IntWritable.class,
  SequenceFileOutputFormat.class);
```

In addition to the `saveAsHadoopFile` and `saveAsNewAPIHadoopFile` functions, if you want more control over writing out a Hadoop format you can use `saveAsHadoopDataset` / `saveAsNewAPIHadoopDataset`. Both functions just take a configuration object on which you need to set all of the Hadoop properties. The configuration is done the same as one would do for configuring the output of a Hadoop MapReduce job.

Protocol Buffers

Protocol buffers [1] were first developed at Google for internal RPCs and have since been open sourced. Protocol Buffers (PB) are structured data, with the fields and types of fields being clearly defined. Protocol Buffers are optimized to be fast for encoding and decoding and also take up the minimum amount of space. Compared to XML protocol buffers are 3x to 10x smaller and can be 20x to 100x faster to encode and decode. While a PB has a consistent encoding there are multiple ways to create a file consisting of many PB messages.

Protocol Buffers are defined using a domain specific language and then the protocol buffer compiler can be used to generate accessor methods in a variety of languages (including all those supported by Spark). Since protocol buffers aim to take up a minimal

1. sometimes called pbs or protobufs

amount of space they are not "self-describing" as encoding the description of the data would take up additional space. This means that to parse data which is formatted as PB we need the protocol buffer definition to make sense of data.

Protocol buffers consist of fields which can be either optional, required, or repeated. When parsing data, a mission optional field does not result in a failure, but a missing required field results in failing to parse the data. As such when adding new fields to existing protocol buffers it is good practice to make the new fields optional as not everyone will upgrade at the same time (and even if they do you might want to read your old data).

Protocol buffers fields can be many pre-defined types, or another protocol buffer message. These types include string, int32, enums, and more. This is by no mean a complete introduction to protocol buffers, if you are interested you should consult the protobuf website. For our example we will look at loading many VenueResponse objects from our sample proto.

Example 5-26. Sample protocol buffer definition

```
message Venue {
  required int32 id = 1;
  required string name = 2;
  required VenueType type = 3;
  optional string address = 4;

  enum VenueType {
    COFFEESHOP = 0;
    WORKPLACE = 1;
    CLUB = 2;
    OMNOMNOM = 3;
    OTHER = 4;
  }
}

message VenueResponse {
  repeated Venue results = 1;
}
```

Twitter's Elephant Bird library, that we used in the previous section to load JSON data, also supports loading and saving data from protocol buffers. Lets look at writing out some Venues.

Example 5-27. Scala Elephant Bird Protocol buffer write out example

```
val job = new Job()
val conf = job.getConfiguration
LzoProtobufBlockOutputFormat.setClassConf(classOf[Places.Venue], conf);
val dnaLounge = Places.Venue.newBuilder()
dnaLounge.setId(1);
dnaLounge.setName("DNA Lounge")
```

```
dnaLounge.setType(Places.Venue.VenueType.CLUB)
val data = sc.parallelize(List(dnaLounge.build()))
val outputData = data.map{ pb =>
  val protoWritable = ProtobufWritable.newInstance(classOf[Places.Venue]);
  protoWritable.set(pb)
  (null, protoWritable)
}
outputData.saveAsNewAPIHadoopFile(outputFile, classOf[Text],
  classOf[ProtobufWritable[Places.Venue]],
  classOf[LzoProtobufBlockOutputFormat[ProtobufWritable[Places.Venue]]], conf)
```

When building your project make sure to use the same protocol buffer library version as Spark, as of this writing that is version 2.5

File Systems

Spark supports a large number of file systems for reading and writing to which we can use with any of the file formats we want.

Local/"Regular" FS

While Spark supports loading files from the local file system, it can be somewhat less convenient to work with compared to the other options. While it doesn't require any setup, it **requires that the files are available on all the nodes in your cluster**.

Some network file systems, like NFS, AFS, MapR's NFS layer, are exposed to the user as a regular file system. If your data is already in one of these forms, then you can use it as an input by just specifying `pathtofile` as the path and Spark will handle it.

Example 5-28. Scala load compressed text file from local FS

```
val rdd = sc.textFile("file:///home/holden/happypandas.gz")
```

If the file isn't already on all nodes in the cluster, you can load it locally and then call `parallelize`. We can also use `addFile(path)` to distribute the contents and then use `SparkFiles.get(path)` in place where we would normally specify the location (e.g. `sc.textFile(SparkFiles.get(…))`). Both of these approaches can be slow, so consider if you can put your files in HDFS, on S3 or similar.

Amazon S3

S3 is an increasingly popular option for storing large amount of data. S3 is especially fast when our compute nodes are located inside of EC2, but can easily have much worse performance if we have to go over the public internet.

Spark will check the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables for your S3 credentials.

HDFS

The Hadoop Distributed File System (HDFS) is a popular distributed file system that Spark works well with. HDFS is designed to work on commodity hardware and be resilient to node failure while providing high data throughput. Spark and HDFS can be collocated on the same machines and Spark can take advantage of this data locality to avoid network overhead.

Using Spark with HDFS is as simple as specifying `hdfs://master:port/path` for your input and output. The Deploying Spark chapter covers how to setup spark for HDFS systems requiring authentication.

The HDFS protocol has changed, if you run a version of Spark which is compiled for a different version it will fail. By default Spark is built against 1.0.4, if you build from source you can specify `SPARK_HADOOP_VERSION=` as a environment variable to build against a different version or you can download a different precompiled version of Spark. The value can be determined by running `hadoop version`.

Compression

Frequently when working with big data, we find our selves needing to use compressed data to save storage space and network overhead. With most Hadoop output formats we can specify a compression codec which will compress the data. As we have already seen, Spark's native input formats (textFile and SequenceFile) can automatically handle some types of compression for us. When reading in compressed data, there are some compression codecs which can be used to automatically guess the compression type.

These compression options only apply to the Hadoop formats which support compression, namely those which are written out to a file system. The database Hadoop formats generally do not implement support for compression, or if they have compressed records it is configured in the database its self rather than with the Hadoop connector.

Choosing an output compression codec can have a big impact on future users of the data. With distributed systems such as Spark we normally try and read our data in from multiple difference machines. To make this possible each worker needs to be able to

find the start of a new record. Some compression formats make this impossible, which requires a single node read in all of the data which can easily lead to a bottleneck. Formats which can be easily read from multiple machines are called "splittable".

Table 5-3. Compression Options

Format	Splittable	Average Compression Speed	Effectiveness on Text	Hadoop Compression Codec	Pure Java	Native	Comments
gzip	N	Fast	High	`org.apache.hadoop.io.compress.GzipCodec`	Y	Y	
lzo	Y [a]	Very fast	Medium	`com.hadoop.compression.lzo.LzoCodec`	Y	Y	LZO require installation on every worker node
bzip2	Y	Slow	Very high	`org.apache.hadoop.io.compress.BZip2Codec`	Y	Y	Uses pure Java for splittable version
zlib	N	Slow	Medium	`org.apache.hadoop.io.compress.DefaultCodec`	Y	Y	Default compression codec for Hadoop
Snappy	N	Very Fast	Low	`org.apache.hadoop.io.compress.SnappyCodec`	N	Y	There is a pure java port of Snappy but it is not currently available in Spark/ Haddop

[a] Depends on the library used

While Spark's textFile method *can* handle compressed input, they automatically disable `splittable` even if the input is compressed in such a way that it could be read in a splittable way. If you find your self needing to read in a large single file compressed input, you should consider skipping Spark's wrapper and instead use either `newAPIHadoopFile` or `hadoopFile` and specify the correct compression codec.

Some data formats (like Sequence files) allow us to only compress the data of our key value data, which can be useful for doing lookups. Other data formats have their own compression control, for example many of the formats in twitter's Elephant Bird package work with LZO compressed data.

Spark wraps both the old and new style APIs for specifying the compression codec.

If we don't know what the compression format is while writing our code, we can instead use the `CompressionCodecFactory` to determine the codec based on the file name.

Spark SQL

When loading data with Spark SQL, the resulting RDDs consist of Row objects. In Java and Scala the Row objects we get back allow access based on the column number. Each Row object has a `get` method that gives back a general type we can cast, and specific get methods for common basic types (e.g. `getFloat`, `getInt`, `getLong`, `getString`, `getShort`, `getBoolean`). In Python you can just access the elements with `row[column_number]` and `row.column_name`. We will cover this in detail the Spark SQL Chapter.

Spark SQL is the simplest way to load data from JSON, Hive, Parquet, and many other semi-structured and structured formats. We will cover how to load these different formats in (to come). For now we will examine how to load JSON data with Spark SQL.

JSON

Using schema inference, Spark SQL can load JSON data. Before we can start to load our data, we need to create an instance of the `SQLContext`.

To load our JSON data all we need to do is call the `jsonFile` function on our `sqlCtx`.

Example 5-29. Python Load JSON with Spark SQL

```
input = sqlCtx.jsonFile(inputFile)
```

Example 5-30. Scala Load JSON with Spark SQL

```
val input = sqlCtx.jsonFile(inputFile)
```

Example 5-31. Java Load JSON with Spark SQL

```
JavaSchemaRDD input = sqlCtx.jsonFile(jsonFile);
```

As part of schema inference, Spark SQL determine types of the different fields. This can be especially useful with large amounts of JSON data where we aren't certain of the schema. We illustrate how to access this information in (to come).

Databases

Spark loads and writes data with database, and database like systems in two primary ways, Spark SQL and Hadoop connectors.

Elasticsearch

Spark can both read and write data from Elasticsearch using ElasticSearch-Hadoop. Elasticsearch is a new open source Lucene based search system. Most of the connectors we have looked at so far have written out to files, this connector instead wraps RPCs to the Elasticsearch cluster.

The elastic search connector is a bit different than the other connectors we have examined, since it ignores the path information we provide instead depends on setting up configuration on our Spark context. The Elasticsearch OutputFormat connector also doesn't quite have the types to use Spark's wrappers, so we instead use `saveAsHadoopDataSet` which means we need to set more properties by hand. Lets look at how to read/write some simple data out to Elastic Search.

The latest ElasticSearch Spark connector is even easier to use, supporting returning Spark SQL rows. This connector is still covered as the row conversion doesn't yet support all of the native types in ElastSearch.

Example 5-32. Scala Elastic Search Output Example

```
val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set("mapred.output.format.class", "org.elasticsearch.hadoop.mr.EsOutputFormat")
jobConf.setOutputCommitter(classOf[FileOutputCommitter])
jobConf.set(ConfigurationOptions.ES_RESOURCE_WRITE, "twitter/tweets")
jobConf.set(ConfigurationOptions.ES_NODES, "localhost")
FileOutputFormat.setOutputPath(jobConf, new Path("-"))
output.saveAsHadoopDataset(jobConf)
```

Example 5-33. Scala Elastic Search Input Example

```
def mapWritableToInput(in: MapWritable): Map[String, String] = {
  in.map{case (k, v) => (k.toString, v.toString)}.toMap
}

val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set(ConfigurationOptions.ES_RESOURCE_READ, args(1))
jobConf.set(ConfigurationOptions.ES_NODES, args(2))
val currentTweets = sc.hadoopRDD(jobConf,
  classOf[EsInputFormat[Object, MapWritable]], classOf[Object],
  classOf[MapWritable])
// Extract only the map
// Convert the MapWritable[Text, Text] to Map[String, String]
val tweets = currentTweets.map{ case (key, value) => mapWritableToInput(value) }
```

Compared to some of our other connectors this is a bit convoluted, but serves as a useful reference for how to work with these types of connectors.

On the write side Elasticsearch can do mapping inference, but this can occasionally infer the types incorrectly, so it can be a good idea to explicitly set a mapping if you are storing things besides strings.

Java Database Connectivity (JDBC)

In addition to using Hadoop input formats, you can create RDDs from JDBC queries. Unlike the other methods of loading data, rather than calling a method on the `SparkContext` we instead create an instance of `org.apache.spark.rdd.JdbcRDD` and provide it with our SparkContext and the other input data it requires.

We will create a simple `JdbcRDD` using a MySQL DB as our data source.

Example 5-34. Scala JdbcRDD Example

```
def createConnection() = {
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  DriverManager.getConnection("jdbc:mysql://localhost/test?user=holden");
}

def extractValues(r: ResultSet) = {
  (r.getInt(1), r.getString(2))
}

val data = new JdbcRDD(sc,
  createConnection, "SELECT * FROM panda WHERE ? <= id AND ID <= ?",
  lowerBound = 1, upperBound = 3, numPartitions = 2, mapRow = extractValues)
println(data.collect().toList)
```

The min and max we provide to the `JdbcRDD` class allow Spark to query different ranges of the data on different machines, so we don't get bottlenecked trying to load all the data on a single node. If you don't know how many records there are, you can just do a count query manually first and use the result.

Along similar lines we provide a function to establish the connection to our database. This lets each node create its own connection to load data over.

The last parameter converts the result from java.sql.ResultSet to a format that is useful for manipulating our data. If left out Spark will automatically convert everything to arrays of objects.

The JdbcRDD is currently accessed a bit differently from most of the other methods for loading data in to Spark, and provides another option for interfacing with database systems.

HBase

Spark ships with examples of how to work with HBase and we will expand on this section in the future. For now if you want to use Spark with HBase, take a look at the HBase examples in Spark.

Cassandra

Spark's Cassandra support has improved greatly with the introduction of the open source Spark Cassandra connector from DataStax. Since the connector is not currently part of Spark, you will need to add some additional dependencies to your build file. Cassandra isn't exactly part of Spark SQL but it returns RDDs of `CassandraRow` objects, which have some of the same methods as Spark SQL's `Row` object.

The Spark Cassandra connector is currently only available in Java and Scala.

Example 5-35. sbt requirements

```
"com.datastax.spark" %% "spark-cassandra-connector" % "1.0.0-rc5",
"com.datastax.spark" %% "spark-cassandra-connector-java" % "1.0.0-rc5"
```

Example 5-36. Maven requirements

```
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector-java</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
```

Much like with Elastic Search, the Cassandra connector reads a job property to determine which cluster to connect to. We set the `spark.cassandra.connection.host` to point to our Cassandra cluster and if we have a username and password we can set them with `spark.cassandra.auth.username` and `spark.cassandra.auth.password`. Assuming you only have a single Cassandra cluster to connect to we can set this up when we are creating our Spark context.

Example 5-37. Scala set Cassandra property

```
val conf = new SparkConf(true)
        .set("spark.cassandra.connection.host", "hostname")

val sc = new SparkContext(conf)
```

Example 5-38. Java set Cassandra property

```
SparkConf conf = new SparkConf(true)
  .set("spark.cassandra.connection.host", cassandraHost);
JavaSparkContext sc = new JavaSparkContext(
  sparkMaster, "basicquerycassandra", conf);
```

The Datastax Cassandra connector uses implicits in Scala to provide some additional functions on top of the SparkContext and RDDs. Lets import the implicit conversions and try loading some data.

Example 5-39. Scala load entire table as an RDD with key/value data

```
// Implicits that add functions to the SparkContext & RDDs.
import com.datastax.spark.connector._
...
// entire table as an RDD
// assumes your table test was created as CREATE TABLE test.kv(key text PRIMARY KEY, value int);
val data = sc.cassandraTable("test" , "kv")
// print some basic stats on the value element
data.map(row => row.getInt("value")).stats()
```

In Java we don't have implicit conversions so we need to explicitly convert our SparkContext and RDDs for this functionality.

Example 5-40. Java load entire table as an RDD with key/value data

```
import com.datastax.spark.connector.CassandraRow;
import static com.datastax.spark.connector.CassandraJavaUtil.javaFunctions;
...
// entire table as an RDD
// assumes your table test was created as CREATE TABLE test.kv(key text PRIMARY KEY, value int);
JavaRDD<CassandraRow> data = javaFunctions(sc).cassandraTable("test" , "kv");
// print some basic stats
System.out.println(data.mapToDouble(new DoubleFunction<CassandraRow>() {
    public double call(CassandraRow row) {
      return row.getInt("value");
      }}).stats());
```

In addition to loading an entire table we can also query subsets of our data. We can restrict our data by adding a where clause tot eh cassandraTable call e.g. `sc.cassandraTable(…).where("key=?", "panda")`.

The Cassandra connector supports saving to Cassandra from a variety of RDD types. We can directly save RDDs of `CassandraRow` objects, which is useful for copying data

between tables. For saving RDDs that aren't in Row form we can save RDDs of tuples and lists by specifying the column mapping.

Example 5-41. Scala save to Cassandra example

```
val rdd = sc.parallelize(List(Seq("moremagic", 1)))
rdd.saveToCassandra("test" , "kv", SomeColumns("key", "value"))
```

This section only briefly introduced the Cassandra connector, for more information check out the connector's github page and our later Spark Streaming chapter.

Conclusion

With the end of this chapter you should be able to get your data into Spark to work with and store the result of your computation in a format that is useful for you. We have examined a number of different formats we can use for our data, as well as compression options and their implications on how data can be consumed. Subsequent chapters will examine ways to write more effective and powerful Spark programs now that we can load and save large data sets.

CHAPTER 6

Advanced Spark Programming

Introduction

This chapter introduces a variety of advanced Spark programming features that we didn't get to cover in the previous chapters. We introduce two types of shared variables, *accumulators* to aggregate information and *broadcast variables* to efficiently distribute large values. Building on our existing transformations on RDDs, we introduce batch operations for tasks with high setup costs, like querying a database. To expand the range of tools accessible to us, we cover Spark's methods for interacting with external programs, such as scripts written in R.

Throughout this chapter we build an example using HAM radio operators' call logs as the input. These logs, at the minimum, include the call signs of the stations contacted. Call signs are assigned by country, and each country has its own range of call signs so we can look up the countries involved. Some call logs also include the physical location of the operators, which we can use to determine the distance involved. We include a sample log entry below. The book's sample repo includes a list of call signs to look up the call logs for and process the results.

Example 6-1. Sample call log entry JSON, with some fields removed

```
{"address":"address here", "band":"40m","callsign":"KK6JLK","city":"SUNNYVALE",
"contactlat":"37.384733","contactlong":"-122.032164",
"county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin",
"id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735",...}
```

The first set of Spark features we look at are shared variables, which are a special type of variable one can use in Spark tasks. In our example we use Spark's shared variables to count non-fatal error conditions and distribute a large lookup table.

When our task involves a large setup time, such as creating a database connection or random number generator, it is useful to share this setup work across multiple data

items. Using a remote call sign lookup database, we examine how to reuse setup work by operating on a per-partition basis.

In addition to the languages directly supported by Spark, the system can call into programs written in other languages. This chapter introduces how to use Spark's language agnostic, `pipe` method to interact with other programs through standard input and output. We illustrate using the `pipe` method to access an `R` library computing the distance of a HAM radio operator's contacts.

Finally, similar to Spark's tools for working with key-value pairs, Spark has methods for working with numeric data. We demonstrate these methods by removing outliers from the distances computed with our HAM radio call logs.

Accumulators

When we normally pass functions to Spark, such as a `map` function or a condition for `filter`, they can use variables defined outside them in the driver program, but each task running on the cluster gets a new copy of each variable, and updates from these copies are not propagated back to the driver. Spark's shared variables, *accumulators* and *broadcast variables*, relax this restriction for two common types of communication patterns: aggregation of results and broadcasts.

Our first type of shared variable, *accumulators*, provide a simple syntax for aggregating values from worker nodes back to the driver program. One of the most common uses of accumulators is to count events that occur during job execution for debugging purposes. For example, say that we are loading a list of all of the call signs we want to retrieve logs for from a file, but we are also interested in how many lines of the input file were blank (perhaps we do not expect to see many such lines in valid input).

Example 6-2. Python Accumulator empty line count example

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines   # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

Example 6-3. Scala Accumulator empty line count example

```
val sc = new SparkContext(...)
val file = sc.textFile("file.txt")

val blankLines = sc.accumulator(0)  // Create an Accumulator[Int] initialized to 0

val callSigns = file.flatMap(line => {
  if (line == "") {
    blankLines += 1 // Add to the accumulator
  }
  line.split(" ")
})

callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

Example 6-4. Java Accumulator empty line count example

```
JavaRDD<String> rdd = sc.textFile(args[1]);

final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
  new FlatMapFunction<String, String>() { public Iterable<String> call(String line) {
      if (line.equals("")) {
        blankLines.add(1);
      }
      return Arrays.asList(line.split(" "));
  }});

callSigns.saveAsTextFile("output.txt")
System.out.println("Blank lines: "+ blankLines.value());
```

In this example, we create an `Accumulator[Int]` called `blankLines`, and then add 1 to it whenever we see a blank line in the input. After evaluation of the transformation, we print the value of the counter. Note that we will only see the right count *after* we run the `saveAsTextFile` action, because the transformation above it, `map`, is lazy, so the side-effect incrementing of the accumlator will only happen when the lazy `map` transformation is forced to occur by the `saveAsTextFile` action.

Of course, it is possible to aggregate values from an entire RDD back to the driver program using actions like `reduce`, but sometimes we need a simple way to aggregate values that, in the process of transforming an RDD, are generated at different scale or granularity than that of the RDD itself. In the previous example, accumulators let us count errors as we load the data, without doing a separate `filter` or `reduce`.

To summarize, accumulators work as follows:

- They are created in the driver by calling the `SparkContext.accumulator(initialValue)` method, which produces an accumulator holding an initial value. The return type is a `spark.Accumulator[T]` object where `T` is the type of `initialValue`.
- Worker code in Spark closures can add to the accumulator with its `+=` method (or `add` in Java).
- The driver program can call `value` on the accumulator to access its value (or call `value()` and `setValue()` in Java).

Note that tasks on worker nodes cannot access the accumulator's `value` — from the point of view of these tasks, accumulators are *write-only* variables. This allows accumulators to be implemented efficiently, without having to communicate every update.

The type of counting shown here becomes especially handy when there are multiple values to keep track of, or when the same value needs to increase at multiple places in the parallel program (for example, you might be counting calls to a JSON parsing library throughout your program). For instance, often we expect some percentage of our data to be corrupted, or allow for the backend to fail some number of times. To prevent producing garbage output when there are too many errors, we can use a counter for valid records and a counter for invalid records. The value of our accumulators is only available in the driver program so that is where we place our checks.

Continuing from our last example, we can now validate the call signs and write the output only if most of the input is valid. The HAM radio call sign format is specified in Article 19 by the International Telecommunication Union, from which we construct a regular expression to verify conformance.

Example 6-5. Python Accumulator error count example

```
# Create Accumulators for validating call signs
validSignCount = sc.accumulator(0)
invalidSignCount = sc.accumulator(0)

def validateSign(sign):
    global validSignCount, invalidSignCount
    if re.match(r"\A\d?[a-zA-Z]{1,2}\d{1,4}[a-zA-Z]{1,3}\Z", sign):
        validSignCount += 1
        return True
    else:
        invalidSignCount += 1
        return False

# Count the number of times we contacted each call sign
validSigns = callSigns.filter(validateSign)
contactCount = validSigns.map(lambda sign: (sign, 1)).reduceByKey(lambda (x, y): x + y)

# Force evaluation so the counters are populated
contactCount.count()
```

```
if invalidSignCount.value < 0.1 * validSignCount.value:
    contactCount.saveAsTextFile(outputDir + "/contactCount")
else:
    print "Too many errors: %d in %d" % (invalidSignCount.value, validSignCount.value)
```

Accumulators and Fault Tolerance

Spark automatically deals with failed or slow machines by re-executing failed or slow tasks. For example, if the node running a partition of a `map` operation crashes, Spark will rerun it on another node; and even if the node does not crash, but is simply much slower than other nodes, Spark can preemptivley launch a "speculative" copy of the task on another node, and take its result if that finishes. Even if no nodes fail, Spark may have to rerun a task to rebuild a cached value that falls out of memory. The net result is therefore that the same function may run multiple times on the same data depending on what happens on the cluster.

How does this interact with accumulators? The end result is that *for accumulators used in actions, Spark only applies each task's update to each accumulator once*. Thus if we want a reliable absolute value counter, regardless of failures or multiple evaluations, we must put it inside an action like `foreach`.

For accumulators used in RDD transformations instead of actions, this guarantee does not exist. An accumulator update within a transformation can occur more than once. One such case of a probably unintended multiple update occurs when a cached but infrequently used RDD is first evicted from the LRU cache and is then subsequently needed. This forces the RDD to be recalculated from its lineage, with the unintended side-effect that calls to update an accumulator within the transformations in that lineage are sent again to the driver. Within transformations, accumulators should, consequently, only be used for debugging purposes.

While future versions of Spark may change this behavior to only count the update once, the current, 1.0.0, version does have the multiple update behavior, so accumulators in transformations are recommended only for debugging purposes.

Custom Accumulators

So far we've seen how to use one of Spark's built-in accumulator types: integers (`Accumulator[Int]`) with addition. Out of the box, Spark also supports accumulators of type `Double`, `Long`, and `Float`. In addition to these, Spark also includes an API to define custom accumulator types and custom aggregation operations (e.g., maximum of the accumulated values instead of addition of them). Custom accumulators need to extend AccumulatorParam which is covered in the Spark API documentation. We can make accumulators with any operation provided it is commutative and associative.

An operation *op* is commutative if *a op b = b op a* for all values *a*, *b*.

An operation *op* is associative if *(a op b) op c = a op (b op c)* for all values *a*, *b* and *c*.

For example, sum and max are commutative and associative operations that are commonly used in Spark accumulators.

Broadcast Variables

Spark's second type of shared variable, *broadcast variables*, allow the program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. They come in handy, for example, if your application needs to send a large read-only lookup table to all the nodes, or even a large feature vector in a machine learning algorithm.

Recall that Spark automatically sends all variables referenced in your closures to the worker nodes. While this is convenient, it can also be inefficient because (1) the default task launching mechanism is optimized for small task sizes, and (2) you might, in fact, use the same variable in *multiple* parallel operations, but Spark will send it separately for each operation. As an example, say that we wanted to write a Spark program that does country lookup our call signs by prefix matching in an array. This is useful for HAM radio call signs since each country gets its own prefix, although the prefixes are not uniform in length. If we wrote this naively in Spark, the code might look like this:

Example 6-6. Python country lookup example

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

This program would run, but, if we had a larger table (like with IP addresses instead of call signs), the `signPrefixes` could easily be several megabytes in size, making it expensive to send that `Array` from the master alongside each task. In addition, if we used the same `signPrefixes` object later (maybe we next ran the same code on `file2.txt`), it would be sent *again* to each node.

We can fix this by making `signPrefixes` a broadcast variable. A broadcast variable is simply an object of type `spark.broadcast.Broadcast[T]`, which wraps a value of type `T`. We can access this value by calling `value` on the Broadcast object in our tasks. The value is sent to each node only once, using an efficient, BitTorrent-like communication mechanism.

Using broadcast variables, our previous example looks like this:

Example 6-7. Python country lookup example with Broadcast values

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))

countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Example 6-8. Scala country lookup example with Broadcast values

```
// Lookup the countries for each call sign for the
// contactCounts RDD.  We load an array of call sign
// prefixes to country code to support this lookup.
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
  val country = lookupInArray(sign, signPrefixes.value)
  (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Example 6-9. Java country lookup example with Broadcast values

```
// Read in the call sign table
// Lookup the countries for each call sign in the
// contactCounts RDD
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable());
JavaPairRDD<String, Integer> countryContactCounts = contactCounts.mapToPair(
  new PairFunction<Tuple2<String, Integer>, String, Integer> (){
    public Tuple2<String, Integer> call(Tuple2<String, Integer> callSignCount) {
      String sign = callSignCount._1();
      String country = lookupCountry(sign, callSignInfo.value());
      return new Tuple2(country, callSignCount._2());
    }}).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

As shown above, broadcast variables are simple to use:

- Create a `Broadcast[T]` by calling `SparkContext.broadcast` on an object of type `T`. Any type works as long as it is also `Serializable`.
- Access its value with the `value` property (or `value()` method in Java).
- The variable will be sent to each node only once, and should be treated as read-only (updates will *not* be propagated to other nodes).

The easiest way to satisfy the *read-only* requirement is to broadcast a primitive value or a reference to an immutable object. In such cases, you won't be able to change the value of the broadcast variable except within the driver code. However, sometimes it can be more convenient or more efficient to broadcast a mutable object. If you do that, then it is up to you to maintain the read-only condition. As we did with our call sign prefix table, of `Array[String]`, then it is up to us to make sure that our code that we run on your worker nodes does not try to do something like `val theArray = broadcastArray.value; theArray(0) = newValue`. When run in a worker node, that line will assign `newValue` to the first array element only in the copy of the array local to the worker node running the code; it will not change the contents of `broadcastArray.value` on any of the other worker nodes.

Optimizing Broadcasts

When broadcasting large values, it is important to choose a data serialization format that is both fast and compact, because the time to send the value over the network can quickly become a bottleneck if it either takes a long time to serialize a value or it takes a long time to send the serialized value over the network. In particular, Java Serialization, the default serialization library used in Spark's Scala and Java APIs, can be very inefficient out of the box for anything except arrays of primitive types. You can optimize serialization by selecting a different serialization library using the `spark.serializer` property (the chapter on simple optimizations will describe how to use *Kryo*, a faster serialization library), or by implementing your own serialization routines for your data types (e.g. using the `java.io.Externalizable` interface for Java serialization, or using the *`reduce`* method to define custom serialization for Python's pickle library).

Working on a Per-Partition Basis

Working with data on a per-partition basis allows us to avoid redoing setup work for each data item. Operations like opening a database connection or creating a random number generator are examples of setup steps that we wish to avoid doing for each element. Spark has *per-partition* versions of `map` and `foreach` to help reduce the cost of these operations, by letting you run code only once for each partition of an RDD.

Going back to our example with call signs, there is an on-line database of HAM radio call signs we can query for a public list of their logged contacts. By using partition based operations, we can share a connection pool to this database to avoid setting up many connections, and re-use our JSON parser. We use the `mapPartitions` function, which gives us an iterator of the elements in each partition of the input RDD and expects us to return an iterator of our results.

Example 6-10. Python shared connection pool

```
def processCallSigns(signs):
    """Lookup call signs using a connection pool"""
    # Create a connection pool
    http = urllib3.PoolManager()
    # the URL associated with each call sign record
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # create the requests (non-blocking)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # fetch the results
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
    # remove any empty results and return
    return filter(lambda x: x[1] is not None, result)

def fetchCallSigns(input):
    """Fetch call signs"""
    return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))

contactsContactList = fetchCallSigns(validSigns)
```

Example 6-11. Scala shared connection pool and JSON parser

```
val contactsContactLists = validSigns.distinct().mapPartitions{
  signs =>
  val mapper = createMapper()
  val client = new HttpClient()
  client.start()
  // create http request
  signs.map {sign =>
    createExchangeForSign(sign)
 // fetch responses
  }.map{ case (sign, exchange) =>
      (sign, readExchangeCallLog(mapper, exchange))
  }.filter(x => x._2 != null) // Remove empty CallLogs
}
```

Example 6-12. Java shared connection pool and JSON parser

```
// Use mapPartitions to re-use setup work.
JavaPairRDD<String, CallLog[]> contactsContactLists = validCallSigns.mapPartitionsToPair(
  new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
    public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input) {
      // List for our results.
      ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>();
```

```
      ArrayList<Tuple2<String, ContentExchange>> requests = new ArrayList<>();
      ObjectMapper mapper = createMapper();
      HttpClient client = new HttpClient();
      try {
        client.start();
        while (input.hasNext()) {
          requests.add(createRequestForSign(input.next(), client));
        }
        for (Tuple2<String, ContentExchange> signExchange : requests) {
          callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
        }
      } catch (Exception e) {
      }
      return callsignLogs;
    }});
System.out.println(StringUtils.join(contactsContactLists.collect(), ","));
```

When operating on a per-partition basis, Spark gives our function an Iterator of the elements in that partition. To return values, we return an Iterable. In addition to `mapPartitions`, Spark has a number of other per-partition operators:

Table 6-1. Per-Partition Operators

Function Name	We are called with	We return
mapPartitions	Iterator of the elements in that partition	Iterator of our return elements
mapPartitionsWithIndex	Integer of partition number, and Iterator of the elements in that partition	Iterator of our return elements
foreachPartition	Iterator of the elements	Nothing

In addition to avoiding setup work, we can sometimes use `mapPartitions` to avoid object creation overhead. Sometimes we need to make an object for aggregating the result which is of a different type. Thinking back to Chapter 3 where we computed the average, one of the ways we did this was by converting our RDD of numbers to an RDD of tuples so we could track the number of elements processed in our reduce step. Instead of doing this for each element, we can instead create the tuple once per partition instead.

Example 6-13. Python average without mapPartitions

```
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])

def basicAvg(nums):
    """Compute the average"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

Example 6-14. Python average with mapPartitions

```
def partitionCtr(nums):
    """Compute sumCounter for partition"""
    sumCount = [0, 0]
```

```
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]

def fastAvg(nums):
    """Compute the avg"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

Piping to External Programs

With three language bindings to chose from out of the box, you may have all the options you need for writing Spark applications. However, if none of Scala, Java or Python does what you need, then Spark provides a general mechanism to pipe data to programs in other languages, like R scripts.

Spark provides a `pipe` method on RDDs. Spark's `pipe` lets us write parts of jobs using any language we want as long as it can read and write to Unix standard streams. With `pipe`, you can write a transformation of an RDD that reads each RDD element from standard input as a String, manipulates that String however you like, and then writes the result(s) as Strings to standard output. The interface and programming model is restrictive and limited, but sometimes it's just what you need to do something like make use of a native code function within a map or filter operation.

Most likely, you'd want to pipe an RDDs content through some external program or script because you've already got a complicated software built and tested that you'd like to reuse with Spark. A lot of data scientist have code in R [1], and we can interact with R programs using `pipe`.

In our example we are going to use an R library to compute the distance for all of the contacts. Each element in our RDD is written out our program with new lines as separators, and every line that the program outputs is a string element in the resulting RDD. To make it easy for our R program to parse the input we will reformat our data to be `mylat, mylon, theirlat, theirlon`. Here we have a comma as the separator.

Example 6-15. R distance program

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) {
  # process line
  contents <- Map(as.numeric, strsplit(line, ","))
```

1. The SparkR project also provides a lightweight front end to use Spark from within R.

```
  mydist <- gdist(contents[[1]][1], contents[[1]][2], contents[[1]][3], contents[[1]][4],
                  units="m", a=6378137.0, b=6356752.3142, verbose = FALSE)
  output = paste(mydist, collapse=" ")
  write(mydist, stdout())
}
```

If that is written to an executable file named `./src/R/finddistance.R`, then it looks like this in use:

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
coffee
NA
ctrl-d
```

So far, so good — we've now got a way to transform every line from `stdin` into output on `stdout`. Now we need to make `finddistance.R` available to each of our worker nodes and to actually transform our RDD with our shell script. To control which separator is used in splitting Both tasks are easy to accomplish in Spark:

Example 6-16. Python driver program using pipe to call finddistance.R

```
# Compute the distance of each call using an external R program
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)
def hasDistInfo(call):
    """Verify that a call has the fields required to compute the distance"""
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))
def formatCall(call):
    """Format a call so that it can be parsed by our R program"""
    return "{0},{1},{2},{3}".format(
        call["mylat"], call["mylong"],
        call["contactlat"], call["contactlong"])

pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

Example 6-17. Scala driver program using pipe to call finddistance.R

```
// Computer the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x => x.map(y =>
  s"$y.contactlay,$y.contactlong,$y.mylat,$y.mylong")).pipe(Seq(
```

```
    SparkFiles.get(distScriptName)))
println(distances.collect().toList)
```

Example 6-18. Java driver program using pipe to call finddistance.R

```
// Computer the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values().map(new VerifyCallLogs()).flatMap(
  new FlatMapFunction<CallLog[], String>() { public Iterable<String> call(CallLog[] calls) {
      ArrayList<String> latLons = new ArrayList<String>();
      for (CallLog call: calls) {
        latLons.add(call.mylat + "," + call.mylong +
                    "," + call.contactlat + "," + call.contactlong);
      }
      return latLons;
    }
  });
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName));
System.out.println(StringUtils.join(distances.collect(), ","));
```

With `SparkContext.addFile(path)`, we can build up a list of files for each of the worker nodes to download with a Spark job. These files can come from the driver's local filesystem (as we did in this example), from HDFS or other Hadoop-supported filesystems, or from an http, https or ftp URI. When an action is run in the job, the files will be downloaded by each of the nodes. The files can then be found on the worker nodes in `SparkFiles.getRootDirectory`, or located with `SparkFiles.get(filename)`. Of course, this is only one way to make sure that `pipe` can find a script on each worker node. You could use another remote copying tool to place the script file in a knowable location on each node.

All the files added with `SparkContext.addFile(path)` are stored in the same directory, so its important to use unique names.

Once the script is available, the `pipe` method on RDDs makes it easy to pipe the elements of an RDD through the script. Perhaps a smarter version of `findDistance` would accept `SEPARATOR` as a command-line argument. In that case, either of these would do the job, although the first is preferred:

- `rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ","))`
- `rdd.pipe(SparkFiles.get("finddistance.R") + " ,")`

In the first option, we are passing the command invocation as a sequence of positional arguments (with the command itself at the zero-offset position); in the second, as a single command string that Spark will then break down into positional arguments.

We can also specify shell environment variables with `pipe` if we desire. Simply pass in a map of environment variable to value as the second parameter to `pipe` and Spark will set those values.

You should now at least have an understanding of how to use `pipe` to process the elements of an RDD through an external command, and of how to distribute such command scripts to the cluster in a way that the worker nodes can find them.

Numeric RDD Operations

Spark provides several descriptive statistics operations on RDDs containing numeric data. These are in addition to the more complex statistical and machine learning methods we will describe later in the Machine Learning chapter.

Spark's numeric operations are implemented with a streaming algorithm which allows for building up our model an element at a time. The descriptive statistics are all computed in a single pass over the data and returned as a `StatsCounter` object by calling `stats()`. The table lists the methods available on the StatsCounter object.

Table 6-2. Summary statistics available from `StatsCounter`

Method	Meaning
count	Number of elements in the RDD
mean	average of the elements
sum	total
max	the maximum value
min	the minimum value
variance	the variance of the elements
sampleVariance	the variance of the elements, computed for a sample
stdev	the standard deviation
sampleStdev	the sample standard deviation

If you only want to compute one of these statistics, you can also call the corresponding method directly on an RDD, e.g., `rdd.mean()` or `rdd.sum()`.

As an example, we will use summary statistics to remove some outliers from our data. Since we will be going over the same RDD twice (once to compute the summary statistics and once to remove the outliers), we may wish to cache the RDD. Going back to our call log example, we can remove the contact points from our call log that are too far away.

Example 6-19. Python remove outliers example

```
# Convert our RDD of strings to numeric data so we can compute stats and
# remove the outliers.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

Example 6-20. Scala remove outliers example

```
// Now we can go ahead and remove outliers since those may have misreported locations
// first we need to take our RDD of strings and turn it into doubles.
val distanceDouble = distance.map(string => string.toDouble)
val stats = distanceDoubles.stats()
val stddev = stats.stdev
val mean = stats.mean
val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3 * stddev)
println(reasonableDistance.collect().toList)
```

Example 6-21. Java remove outliers example

```
// First we need to convert our RDD of String to a DoubleRDD so we can
// access the stats function
JavaDoubleRDD distanceDoubles = distances.mapToDouble(new DoubleFunction<String>() {
    public double call(String value) {
      return Double.parseDouble(value);
    }});
final StatCounter stats = distanceDoubles.stats();
final Double stddev = stats.stdev();
final Double mean = stats.mean();
JavaDoubleRDD reasonableDistances =
  distanceDoubles.filter(new Function<Double, Boolean>() {
    public Boolean call(Double x) {
      return (Math.abs(x-mean) < 3 * stddev);}});
System.out.println(StringUtils.join(reasonableDistance.collect(), ","));
```

With that final piece we have completed our sample application that uses accumulators and broadcast variables, per-partition processing, interfaces with external programs, and summary statistics. The entire source code is available in `./src/python/ChapterSixExample.py`, `src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala` and `src/main/java/com/oreilly/learningsparkexamples/java/ChapterSixExample.java` respectively.

Conclusion

In this chapter, you have been introduced to some of the more advanced Spark programming features that you can use to make your programs more efficient or expressive. Subsequent chapters cover the extension of basic Spark into Spark SQL (similar to Apache Hive) and Spark Streaming (for continuous and real-time data analysis.) We'll also start seeing more complex and more complete sample applications that make use of much of the functionality described so far, and that should help guide and inspire your own usage of Spark. You should have little difficulty expanding your Spark programming repertoire as you proceed.

CHAPTER 7

Running on a Cluster

Introduction

Up to now, we've focused on learning Spark's core API by using the Spark shell and examples that run in Spark's local mode. One benefit of writing applications on Spark is the ability to scale computation by adding more machines and running in cluster mode. The good news is that writing applications for parallel cluster execution uses the same API you've already learned in this book. The examples and applications you've written so far will run on a cluster "out of the box". This is one of the benefits of Spark's higher level API, users can rapidly prototype applications on smaller datasets locally then run unmodified code on even very large clusters.

This chapter first explains the runtime architecture of a distributed Spark application, then discusses options for running Spark in distributed clusters. Spark can run on a wide variety of cluster managers (YARN, Mesos, and its own built-in cluster manager) and in both on-premise and cloud-based deployments. We'll discuss the trade offs and configurations required for running in each case. Along the way we'll also cover the "nuts and bolts" of compiling, deploying, and configuring a Spark application. After reading this chapter you'll have everything you need to build and launch a distributed Spark application. The following chapter will cover tuning and debugging applications.

Spark Runtime Architecture

Before diving into the specifics of running Spark on a cluster, it's helpful to understand the architecture of Spark when running in a cluster.

When running in cluster mode, Spark utilizes a master-slave architecture with one central coordinator and many distributed workers. The central coordinator is called the *driver*. The driver communicates with potentially larger number of distributed workers

called *executors*. The driver runs in its own Java process and each executor is a Java process. A driver and its executors are together termed a Spark *application*.

A Spark application is launched on a set of machines using an external service called a *cluster manager*. Spark is packaged with a built-in cluster manager called the Standalone cluster manager. Spark also works with Apache YARN and Apache Mesos, two popular open source cluster managers.

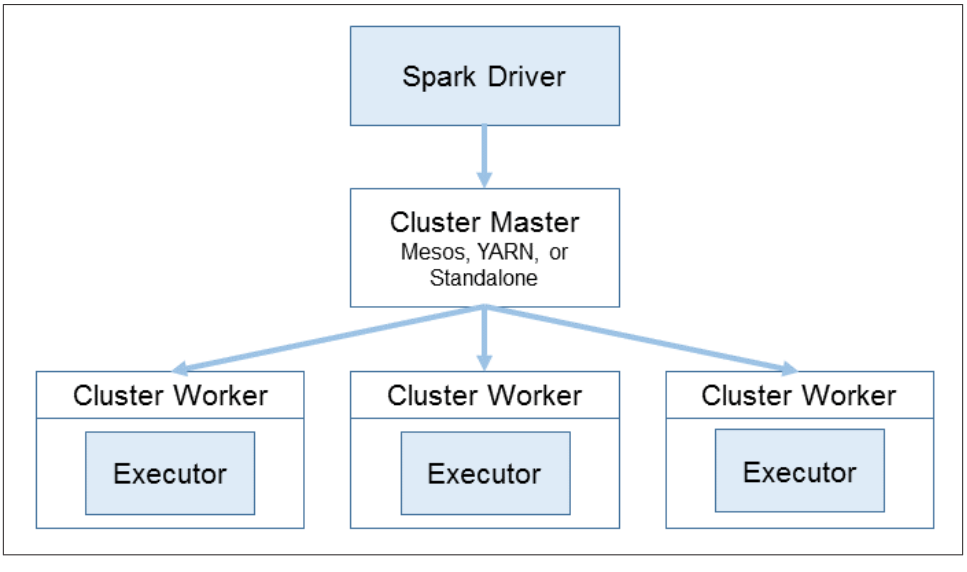

Figure 7-1. The components of a distributed Spark application

The Driver

The driver program it is the process where the `main` method of your program runs. It is the process running the user code that creates a SparkContext, creates RDDs, and performs actions and transformations. When you launch a Spark shell, you've created a driver program (if you remember, the Spark shell comes pre-loaded with a SparkContext called `sc`). Once the driver terminates, the application is finished.

When the driver runs, it performs two duties:

1. Compiling a user program into tasks

A Spark driver is responsible for compiling a user program into units of physical execution called *tasks*. At a high level, all Spark user programs follow the same structure: they create RDDs from some input, derive new RDDs from those using transformations and perform actions to collect or save data. A Spark program implicitly creates a logical direct acyclic graph (DAG) of operations. When the driver runs, it compiles this logical

graph into a physical execution plan. At this time, optimizations such as pipelining are applied to compress the graph into *stages*. Each stage, in turn is decomposed into several parallel *tasks*. The task's are bundled up and prepared to be sent to be executed on the cluster. Tasks are the smallest unit of work in Spark; a typical user program can compile to hundreds or thousands of individual tasks.

2. Scheduling tasks on executors

Given a plan of physical execution, A Spark driver must coordinate the scheduling of individual tasks on executors. When executors are started they register themselves with the driver, so it has a complete view of the application's executors at all time. Each executor represents a process capable of running tasks and storing RDD data.

The Spark driver will look at the current set of executors and try to schedule each task in an appropriate location. When tasks execute, they may have a side-effect of storing cached data. This driver maintains the index of the storage location of every RDD, since those locations may be relevant for scheduling future tasks.

The driver exposes information about the running Spark application through a web interface, by default available at port 4040. This is available even in local mode on the machine where you are running Spark (you can typically browse to `http://localhost:4040`). The specifics of Spark's web UI and scheduling mechanisms are discussed in more detail in an upcoming chapter.

Executors

Spark executors are worker processes responsible for executing the individual tasks of a given Spark job. Executors are launched once at the beginning of a Spark application and they typically survive for the entire lifetime of an application, though Spark applications can continue if executors fail. Executors have two roles. First, they run individual tasks that make up a Spark application. Second, they provide in-memory storage for RDDs that are cached by user programs. Because RDDs are cached directly inside of executors, tasks are able to run alongside cached input data.

Drivers and Executors in Local Mode

For most of this book, you've run examples in Spark's local mode. In this mode, the Spark driver runs along with an executor in the same Java process. This is a special case. Executors typically each run in a dedicated Process.

Cluster manager

So far we've discussed drivers and executors in somewhat abstract terms. But how to drivers and executor processes initially get launched? In cluster mode Spark depends

on a cluster manager to launch executors and, in certain cases, to launch the driver. The cluster manager is a pluggable component in Spark. This design choice allows Spark to run on top of different resource managers such as Apache YARN and Apache Mesos.

Sparks' documentation consistently uses the term *driver* and *executor* when describing the processes that execute a Spark job. The terms *master* and *worker* are used to describe the centralized and distributed portions of the Cluster manager. It's easy to confuse these terms, so pay close attention. For instance Apache YARN runs a master daemon (called the Resource Manager) and several worker daemons called (Node Managers). Spark will run both drivers and executors on YARN worker nodes.

Deploying to a Cluster with Spark-Submit

Spark provides a uniform interface for submitting jobs across all cluster managers, a tool aptly named `spark-submit`. In Chapter 2 you saw a simple example of submitting a Python program with spark-submit:

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

When `spark-submit` is called with nothing but the name of a script or jar, it simply runs the supplied Spark program locally. Let's say we wanted to submit this program to a Spark Standalone cluster. We can provide extra flags with the address of a standalone cluster and a specific size of executor we'd like to launch:

```
bin/spark-submit --master spark://host:7077 --executor-memory 10g my_script.py
```

`spark-submit` provides a variety of options that let you control specific details about a particular run of your application. These options fall roughly into two categories. The first is the location of the cluster manager along with an amount of resources you'd like to request for your job (as shown above). The second is information about the runtime dependencies of your application, such as libraries or files you want to be present on all worker machines.

The general format for spark-submit is the following:

```
bin/spark-submit [options] <app jar | python file> [app options]
```

`[options]` are a list of flags for spark-submit. You can enumerate all possible flags by passing --help to spark-submit. A list of common flags is enumerated in Table 7-1.

`<app jar | python file>` refers to the jar or Python script containing the entry point into your application.

`[app options]` are options that will be passed onto your application. If the `main` method of your program parses its calling arguments, it will see only `[app options]` and not the flags specific to `spark-submit`.

Table 7-1. Common flags for `spark-submit`

Flag	Explanation
`--master`	Indicates the cluster manager to connect to. The options for this flag are described in Table 7-2.
`--deploy-mode`	Whether to launch the driver program locally ("client") or one of the worker machines inside the cluster ("cluster"). In client mode `spark-submit` will run your driver on the same machine where `spark-submit` is itself being invoked. In cluster mode, the driver will be shipped to execute on a worker node in the cluster. The default is client mode.
`--class`	The "main" class of your application if running a Java or Scala program.
`--name`	A human readable name for your application. This will be displayed in Spark's web UI.
`--jars`	A list of jar files to upload and place on the classpath of your application. If your application depends on a small number of third-party jars, you can add them here.
`--files`	A list of files to be placed in the working directory of your application.
`--py-files`	A list of files to be added to the PYTHONPATH of your application.
`--executor-memory`	The amount of memory to use for executors, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 Megabytes) or "15g" (15 Gigabytes).
`--driver-memory`	The amount of memory to use for the driver process, in bytes. Suffixes can be used to specify larger quantities such as "512m" (512 Megabytes) or "15g" (15 Gigabytes).

Table 7-2. Possible values for the `--master` flag in `spark-submit`

Value	Explanation
spark://host:port	Connect to a Spark Standalone master at the specified port. By default Spark Standalone master's listen on port 7077 for submitted jobs.
mesos://host:port	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050 for submitted jobs.
yarn	Indicates submission to YARN cluster. When running on YARN you'll need to export `HADOOP_CONF_DIR` to point the location of your Hadoop configuration directory.
local	Run in local mode with a single core.
local[N]	Run in local mode with N cores.
local[*]	Run in local mode and use as many cores as the machine has.

`spark-submit` also allows setting arbitrary configuration options using either the `--conf prop=value` flag or providing a properties file through --properties-file which contains key-value pairs. A future chapter will discuss Spark's configuration system.

Here are a few longer form example invocations of spark-submit using various options:

```
# Submitting a Java application to standalone cluster mode
$ ./bin/spark-submit \
```

```
  --master spark://hostname:7077 \
  --deploy-mode cluster \
  --class com.databricks.examples.SparkExample \
  --name "Example Program" \
  --jars dep1.jar,dep2.jar,dep3.jar \
  --total-executor-cores 300 \
  --executor-memory 10g \
  myApp.jar "options" "to your application" "go here"

# Submitting a Python application in YARN client mode
$ export HADOOP_CONF_DIR=/opt/hadoop/conf
$ ./bin/spark-submit \
  --master yarn \
  --py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py \
  --deploy-mode client \
  --name "Example Program" \
  --queue exampleQueue \
  --num-executors 40 \
  --executor-memory 10g \
  my_script.py "options" "to your application" "go here"
```

Packaging Your Code and Dependencies

Throughout most of this book we've provided example programs that are self contained and had no library dependencies outside of Spark's API. More often, user programs depend on third party libraries. If your program imports any libraries that are not in org.apache.spark package or part of the language library, you need to take special care that all your dependencies are present at the runtime of your Spark application.

For Python users, there are a few ways to install third party libraries. Since PySpark uses the existing Python installation on worker machines, you can install dependency libraries directly on the cluster machines using standard Python package mangers (such as `pip` or `easy_install`), or via a manual installation into the `site-packages/` directory of your Python installation. Alternatively, you can submit individual libraries using the `--py-files` argument to `spark-submit` and they will be added to the Python interpreter's path. Adding libraries manually is more convenient if you do not have access to install packages on the cluster, but do keep in mind potential conflicts with existing packages already installed on the machines.

What about Spark itself?

When you are bundling an application, you should never include Spark itself in the list of submitted dependencies. When using `spark-submit`, Spark will always be present in the path of your program automatically.

For Java and Scala users, it is also possible to submit individual jar files using the `--jars` flag to `spark-submit`. This can work well if you have a very simple dependency on one or two libraries and they themselves don't have any other dependencies. It is more common, however, for users to have Java or Scala projects that depend on several libraries. When you submit an application to Spark, it must ship with its entire *transitive dependency graph* to the cluster. This means not only the libraries you directly depend on, but their dependencies, their dependencies' dependenices, etc. Manually tracking and submitting this set of jar files would be extremely cumbersome. Instead, it's common practice to rely on a build tool to produce a single large jar containing the entire transitive dependency graph of an application. This is often called an *uber jar* or an *assembly jar*, and most Java or Scala build tools can produce this type of artifact.

Let's look at an example Java project with multiple dependencies that produces an uber jar. Example 7-1 provides an example Maven `pom.xml` file containing a build definition. This example doesn't show the actual Java code or project directory structure, but Maven expects user code to be in a `src/main/java` directory relative to the project root (the root should contain the `pom.xml` file).

Example 7-1. pom.xml file for a Spark application built with Maven build

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <!-- Information about your project -->
  <groupId>com.databricks</groupId>
  <artifactId>example-build</artifactId>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>

  <dependencies>
    <!-- Spark dependency -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- Third party library -->
    <dependency>
      <groupId>net.sf.jopt-simple</groupId>
      <artifactId>jopt-simple</artifactId>
      <version>4.3</version>
    </dependency>
    <!-- Third party library -->
    <dependency>
      <groupId>joda-time</groupId>
      <artifactId>joda-time</artifactId>
      <version>2.0</version>
```

```
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Maven shade plugin which creates uber jars. -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

This project declares two transitive dependencies: `jopt-simple`, a Java library to perform option parsing, and `joda-time`, a library with utilities for time and date conversion. It also depends on Spark, but Spark is marked as `provided` to ensure that Spark is never packaged with the application artifacts. The build includes the `maven-shade-plugin` to create an uber jar containing all if its dependencies. This is enabled by asking Maven to execute the `shade` goal of the plug-in every time a `package` phase occurs. With this build configuration, an uber jar is created automatically when `mvn package` is run.

```
$ mvn package
# In the target directory, we'll see an uber jar and the original package jar
$ ls target/
example-build-1.0.jar
original-example-build-1.0.jar
# Listing the uber jar will reveal classes from dependency libraries
$ jar tf target/example-build-1.0.jar
...
joptsimple/HelpFormatter.class
...
org/joda/time/tz/UTCProvider.class
...
# An uber jar can be passed directly to spark-submit
$ /path/to/spark/bin/spark-submit --master local ... target/example-build-1.0.jar
```

One occasionally disruptive issue when submitting Spark applications that contain dependencies is dealing with dependency conflicts in cases where a user application and Spark itself both depend on the same library. This comes up relatively rarely, but when it does, it can be vexing for users. Typically, this will manifest itself when a `NoSuchMe`

thodError, a ClassNotFoundException, or some other JVM exception related to class loading, is thrown during the execution of a Spark job. There are two solutions to this problem. The first is to modify your application to depend on the same version of the third party library that Spark does. The second is to modify the packaging of your application using a procedure that is often called "shading". The Maven build tool supports shading through advanced configuration of the plug-in shown in the above example (in fact the shading capability is why the plug-in is named maven-shade-plugin). Shading allows you to make a second copy of the conflicting package under a different namespace and re-writes your application's code to use the renamed version. This somewhat brute-force technique is quite effective at resolving runtime dependency conflicts. For specific instructions on how to shade dependencies, see the documentation for your build tool.

Summary of Program Execution

To summarize the concepts in the previous sections, let's walk through the exact steps which occur when you run a Spark application on a cluster.

1. The user submits an application using the spark-submit tool.
2. The spark-submit tool launches the driver program and invokes the main method specified by the user. If the program has been launched in client mode, spark-submit will launch the driver program on the submitting machine. If the program is running in cluster mode, the driver program will be shipped to the cluster and launched on a managed worker node of the cluster.
3. The driver program contacts the cluster manager to ask for resources to launch executors.
4. The cluster manager launches executors on behalf of the driver program.
5. The driver process begins stepping through each line in the user application. Based on the RDD actions and transformations in the program, it will create work for executors in the form of tasks.
6. Tasks are run on executor processes to complete the user's application.
7. If the driver encounters an sc.stop() statement, it will gracefully terminate the executors and release resources from the cluster manager.
8. When the driver's main method exist, it will shut down. Any remaining executors will be terminated on shut down.

Scheduling Within and Between Spark Applications

The example we just walked through involves a single user submitting a job to a cluster. In reality, many clusters are shared between multiple users. Shared environments have

the challenge of scheduling - what happens if two users both launch Spark applications that each want to use the entire cluster's worth of resources? Scheduling policies help ensure that resources are not overwhelmed and allow for prioritization of workloads.

For scheduling in multi-tenant clusters, Spark primarily relies on the cluster manager to share resources between Spark applications. When a Spark application asks for executors from the cluster manager, it may receive more or fewer executors depending on availability and contention in the cluster. Many cluster managers offer the ability to define queues with different priorities or capacity limits and Spark will then submit jobs to such queues. See the documentation of your specific cluster manager for more details.

One special case of Spark applications are those which are *long lived*, meaning that they are never intended to terminate. An example of a long lived Spark application is the JDBC server bundled with Spark SQL. When the JDBC server launches it acquires a set of executors from the cluster manager, then it acts as a permanent gateway for SQL queries submitted by users. Since this single application is scheduling work for multiple users, it needs a finer grained mechanism to enforce sharing policies. Spark provides such a mechanism through configurable intra-application scheduling policies. Spark's internal *Fair Scheduler* lets long lived applications define queues for prioritizing scheduling of tasks. A detailed review of these is beyond the scope of this book; the official documentation on the Fair Scheduler [1] provides a good reference.

Cluster Managers

Spark can run over a variety of *cluster managers* to access the machines in a cluster. If you only want to run Spark by itself on a set of machines, the built-in "standalone mode" is the easiest way to deploy it. However, if you have a cluster that you'd like to share with other distributed applications (e.g. both Spark jobs and Hadoop MapReduce jobs), Spark can also run over two popular cluster managers: Hadoop YARN and Apache Mesos. Finally, for deploying on Amazon EC2, Spark comes with built-in scripts that launch a standalone mode cluster and a variety of supporting services. In this section, we'll cover how to run Spark in each of these environments.

Standalone Mode

Spark's standalone mode offers a simple way to run applications on a cluster. It consists of a *master* and multiple *workers*, each with a configured amount of memory and CPU cores. When you submit an application, you can choose how much memory its executors will use, as well as the total number of cores.

1. *http://spark.apache.org/docs/latest/job-scheduling.html*

Launching the Standalone Cluster Manager

You can start the standalone cluster manager either by starting a master and workers by hand, or by using launch scripts in Spark's `sbin` directory. The launch scripts are the simplest way to use, but require SSH access between your machines and are currently (as of Spark 1.1) only available on Mac OS X and Linux. We will cover these first, then show how to launch a cluster manually on other platforms.

To use the cluster launch scripts, follow the steps below:

1. Copy a compiled version of Spark to the same location on all your machines — for example, `/home/yourname/spark`.
2. Set up password-less SSH access from your master machine to the others. This requires having the same user account on all the machines, creating a private SSH key for it on the master via `ssh-keygen`, and adding this key to the `.ssh/authorized_keys` file of all the workers. If you have not set this up before, you can follow the commands below:

```
# On master: run ssh-keygen accepting default options
$ ssh-keygen -t dsa
Enter file in which to save the key (/home/you/.ssh/id_dsa): [ENTER]
Enter passphrase (empty for no passphrase): [EMPTY]
Enter same passphrase again: [EMPTY]

# On workers: copy ~/.ssh/id_dsa.pub from your master to the worker, then use:
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ chmod 644 ~/.ssh/authorized_keys
```

1. Edit the `$SPARK_HOME/conf/slaves` file on your master to contain the workers' hostnames.
2. To start the cluster, run `$SPARK_HOME/sbin/start-all.sh` on your master (it is important to run it there rather than on a worker). If everything started, you should get no prompts for a password, and the cluster manager's web UI should appear at `http://masternode:8080` and should show all your workers.
3. To stop the cluster, run `$SPARK_HOME/sbin/stop-all.sh` on your master node.

If you are not on a UNIX system or would like to launch cluster manually, you can also start the master and workers by hand, using the `spark-class` script in `$SPARK_HOME/bin`. On your master, type:

```
bin/spark-class org.apache.spark.deploy.master.Master
```

Then on workers:

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://masternode:7070
```

(where `masternode` is the hostname of your master). On Windows, use `\` instead of `/`.

By default, the cluster manager will automatically allocate the amount of CPUs and memory on each worker and pick a suitable default to use for Spark. More details on configuring the standalone manager are available in Spark's official documentation [2].

Submitting Applications

To submit an application to the standalone cluster manager, pass `spark://masternode:7070` as the master argument to `spark-submit`. For example:

```
spark-submit --master spark://masternode:7070 yourapp
```

This cluster URL is also shown in the standalone cluster manager's web UI, at `http://masternode:8080`. Some administrators might configure Spark to use a different port than 7070, so you can always find the URL there.

You can also launch `spark-shell` or `pyspark` against the cluster in the same way, by passing the `--master` parameter:

```
spark-shell --master spark://masternode:7070
pyspark --master spark://masternode:7070
```

To check that your application or shell is running, look at the cluster manager's web UI `http://masternode:8080` and make sure that (1) your application is connected (i.e. it appears under Running Applications) and (2) it is listed as having more than 0 cores and memory.

A common pitfall that might prevent your application from running is requesting more memory for executors (with the `--executor-memory` flag to `spark-submit`) than is available on cluster nodes. In this case, the standalone cluster manager will never give it nodes. Make sure that the value your application is requesting can be satisfied by the cluster.

Finally, the standalone cluster manager supports two *deploy modes* for where the driver program of your application runs. In `client` mode (the default), the driver runs on the machine where you executed `spark-submit`, as part of the `spark-submit` command. This means that you can directly see the output of your driver program, or send input to it (e.g. for an interactive shell), but it requires that the machine where you submitted from has fast connectivity to the workers and stays available for the duration of your application. In contrast, in `cluster` mode, the driver is launched within the standalone cluster, as another process on one of the worker nodes, and then it connects back to request executors. This makes `spark-submit` more "fire-and-forget", in that you can close your laptop while the application is running. You will still be able to access logs

2. *http://spark.apache.org/docs/latest/spark-standalone.html*

for the application through the cluster manager's web UI. You can switch to cluster mode by passing `--deploy-mode cluster` to `spark-submit`.

Configuring Resource Usage

When sharing a Spark cluster among multiple applications, you will need to decide how to allocate resources between them. The standalone cluster manager has a basic scheduling policy that allows capping the usage of each application so that multiple ones run concurrently. Apache Mesos supports more dynamic sharing while an application is running, while YARN has a concept of queues that allows you to cap usage for various sets of applications.

In the standalone cluster manager, resource allocation is controlled by two settings:

- The memory per executor for the application, which you set through the `--executor-memory` argument to `spark-submit`. Each application will have at most one executor on each worker, so this setting controls how much of that worker's memory the application will claim. By default, this setting is 1 GB — you will likely want to increase it on most servers.
- The maximum number of CPU cores the application will claim. By default, this is unlimited, i.e., the application will launch executors on every available node in the cluster. For a multi-user workload, you should instead ask users to cap their usage. You can set this value through the `--total-executor-cores` argument to `spark-submit`, or by configuring `spark.cores.max` in your Spark configuration file.

You can always see the current resource allocation in the standalone manager's web UI, `http://masternode:8080`.

Finally, the standalone cluster manager works by spreading out each application across the maximum number of executors by default. For example, suppose that you have a 20-node cluster with 4-core machines, and you submit an application with `--executor-memory 1G` and `--total-executor-cores 8`. Then Spark will launch 8 executors, each with 1 GB of RAM, on different machines. Spark does this by default to give applications a chance to achieve data locality for distributed file systems running on the same machines (e.g. HDFS), because these systems typically have data spread out across all nodes. If you prefer, you can instead ask Spark to consolidate executors on as few nodes as possible, by setting the config property `spark.deploy.spreadOut` to `false` in `conf/spark-defaults.conf`. In this case, the application above would get only 2 executors, each with 1 GB RAM and 4 cores. This setting affects *all* applications on the standalone cluster and must be configured before you launch the standalone cluster manager.

High Availability

When running in production settings, you will want your standalone cluster to be available to accept applications even if individual nodes in your cluster goes down. The standalone cluster manager, and Spark, are both tolerant of worker nodes going down, even during the computation. If you also want the master of the cluster to be highly available, Spark supports using Apache ZooKeeper (a distributed coordination system) to keep multiple standby masters and switch to a new one when any of them fails. Configuring Spark with ZooKeeper is outside the scope of the book, but is described in the official Spark documentation [3].

Hadoop YARN

YARN is a cluster manager introduced in Hadoop 2.0 that allows diverse data processing frameworks to run on a shared resource pool, and is typically installed on the same nodes as the Hadoop file system (HDFS). Running Spark on YARN in these environments is useful because it lets Spark access HDFS data quickly, on the same nodes where the data is stored.

Using YARN in Spark is straightforward: you set an environment variable that points to your Hadoop configuration directory, then submit jobs to a special master URL with `spark-submit`.

The first step is to figure out your Hadoop configuration directory, and set it as the environment variable `HADOOP_CONF_DIR`. This is the directory that contains `yarn-site.xml` and other config files — typically, it is `HADOOP_HOME/conf` if you installed Hadoop in `HADOOP_HOME`, or a system path like `/etc/hadoop/conf`. Then, submit your application as follows:

```
export HADOOP_CONF_DIR="..."
spark-submit --master yarn yourapp
```

Like with the standalone cluster manager, there are two modes to connect your application to the cluster: client mode, where the driver program for your application runs on the machine that you submitted it from (e.g. your laptop), and cluster mode, where the driver also runs inside a YARN container. You can set the mode to use via the `--deploy-mode` argument to `spark-submit`.

Spark's interactive shell and `pyspark` both work on YARN as well — simply set `HADOOP_CONF_DIR` and pass `--master yarn` to these applications. Note that these will only run in client mode since they need to obtain input from the user.

3. *https://spark.apache.org/docs/latest/spark-standalone.html#high-availability*

Configuring Resource Usage

When running on YARN, Spark executes a fixed number of executors, which you can set via the `--num-executors` flag to `spark-submit`, `spark-shell`, etc. By default, this is only 2, so you will likely need to increase it. You can also set the memory used by each executor via `--executor-memory` and the number of cores it claims from YARN via `--executor-cores`. On a given amount of hardware resources, Spark will usually run better with a smaller number of larger executors (with multiple cores and more memory), since it can optimize communication within each executor. Note however that some clusters have a limit on the maximum size of an executor (8 GB by default), and will not let you launch larger ones.

Some YARN clusters are configured to schedule applications into multiple "queues" for resource management purposes. Use the `--queue` option to select your queue name.

Finally, further information on configuration options for YARN is available in the official Spark documentation [4].

Apache Mesos

Apache Mesos is a general-purpose cluster manager that can run both analytics workloads and long-running services (e.g. web applications or key-value stores) on a dynamically shared cluster. To use Mesos, pass a `mesos://` URI to `spark-submit`:

```
spark-submit --master mesos://masternode:5050 yourapp
```

Mesos clusters can also be configured to use ZooKeeper to elect a master when running in multi-master node. In this case, use a `mesos://zk://` URI pointing to a list of ZooKeeper nodes. For example, if you have three ZooKeeper nodes (`node1`, `node1` and `node1`), on which ZooKeeper is running on port 2181, use the following URI:

```
mesos://zk://node1:2181/mesos,node2:2181/mesos,node3:2181/mesos
```

Mesos Run Modes

Unlike the other cluster managers, Mesos offers two modes to share resources between executors on the same cluster. In "fine-grained" mode, which is the default, executors scale up and down the number of CPUs they claim from Mesos as they execute tasks, and so a machine running multiple executors can dynamically share CPU resources between them. In "coarse-grained" mode, Spark allocates a fixed number of CPUs to each executor in advance and never releases them until the application ends, even if the executor is not currently running tasks. You can enable coarse-grained mode by passing `--conf spark.mesos.coarse=true` to `spark-submit`.

4. *http://spark.apache.org/docs/latest/submitting-applications.html*

The fine-grained Mesos mode is attractive when multiple users share a cluster to run interactive workloads such as shells, because applications will scale down their number of cores when they're not doing work and still allow other users' programs to use the cluster. The downside, however, is that scheduling tasks through fine-grained mode adds more latency (so very low-latency applications like Spark Streaming may suffer), and that applications may need to wait some amount of time for CPU cores to become free to "ramp up" again when the user types a new command. Note, however, that you can use a mix of run modes in the same Mesos cluster (i.e. some of your Spark jobs might have `spark.mesos.coarse` set to true and some might not).

Client and Cluster Mode

As of Spark 1.1, Spark on Mesos only supports running applications in the "client" deploy mode, that is, with the driver running on the machine that submitted the application. If you would like to run your driver in the Mesos cluster as well, frameworks like Aurora [5] and Chronos [6] allow submitting arbitrary scripts to run on Mesos and monitoring them. You can use one of these to launch the driver for your application.

Configuring Resource Usage

You can control resource usage on Mesos through two parameters to `spark-submit`: `--executor-memory`, to set the memory for each executor, and `--total-executor-cores`, to set the maximum number of CPU cores for the application to claim (across all executors). By default, Spark will launch each executor with as many cores as possible, consolidating the application to the smallest number of executors that give it the desired number of cores. If you do not set `--total-executor-cores`, it will try to use all available cores in the cluster.

Amazon EC2

Spark comes with a built-in script to launch clusters on Amazon EC2. This script launches a set of nodes and then installs the standalone cluster manager on them, so once the cluster is up, you can use it according to the standalone mode instructions above. In addition, the EC2 script sets up supporting services such as HDFS, Tachyon, and Ganglia to monitor your cluster.

The Spark EC2 script is called `spark-ec2`, and is located in the `ec2` folder of your Spark installation. It requires Python 2.6 or higher. You can download Spark and run the EC2 script without compiling Spark beforehand.

5. *http://aurora.incubator.apache.org*
6. *http://airbnb.github.io/chronos*

The EC2 script can manage multiple named *clusters*, identifying them using EC2 security groups. For each cluster, the script will create a security group called `clustername-master` for the master node, and `clustername-slaves` for the workers.

Launching a Cluster

To launch a cluster, you should first create an Amazon Web Services (AWS) account and obtain an access key ID and secret access key. Then export these as environment variables:

```
export AWS_ACCESS_KEY_ID="..."
export AWS_SECRET_ACCESS_KEY="..."
```

In addition, create an EC2 SSH key pair and download its private key file (usually called `keypair.pem`) so that you can SSH into the machines.

Next, run the `launch` command of the `spark-ec2` script, giving it your key pair name, private key file, and a name for the cluster. By default, this will launch a cluster with one master and one slave, using `m1.xlarge` EC2 instances:

```
cd $SPARK_HOME/ec2
./spark-ec2 -k mykeypair -i mykeypair.pem launch mycluster
```

You can also configure the instance types, number of slaves, EC2 region, and other factors using options to `spark-ec2`. For example:

```
# Launch a cluster with 5 slaves of type m3.xlarge
./spark-ec2 -k mykeypair -i mykeypair.pem -s 5 -t m3.xlarge launch mycluster
```

For a full list of options, run `spark-ec2 --help`. Some of the most common ones are:

Table 7-3. Common options to `spark-ec2`

Option	Meaning
`-k KEYPAIR`	Name of key pair to use
`-i IDENTITY_FILE`	Private key file (ending in `.pem`)
`-s NUM_SLAVES`	Number of slave nodes
`-t INSTANCE_TYPE`	Amazon instance type to use
`-r REGION`	Amazon region to use (e.g. `us-west-1`)
`-z ZONE`	Availability zone (e.g. `us-west-1b`)
`--spot-price=PRICE`	Use spot instances at the given spot price (in US dollars)

Once you launch the script, it usually takes about 5 minutes to launch the machines, log into them, and set up Spark.

Logging Into a Cluster

You can log into a cluster by SSHing into its master node with the `.pem` file for your keypair. For convenience, `spark-ec2` provides a `login` command for this purpose:

```
./spark-ec2 -k mykeypair -i mykeypair.pem login mycluster
```

Alternatively, you can find the master's hostname by running

```
./spark-ec2 get-master mycluster
```

Then SSH into it yourself using `ssh -i keypair.pem root@masternode`.

Once you are in the cluster, you can use the Spark installation in `/root/spark` to run programs. This is a standalone cluster installation, with the master URL `spark://masternode:7077`. `spark-defaults.conf` on the cluster will be configured to use it by default if you just launch an application with `spark-submit`. You can also access the cluster's web UI at `http://masternode:8080`.

Note that only programs launched from the cluster will be able to submit jobs to it with `spark-submit`; the firewall rules will prevent external hosts from submitting them for security reasons. To run a pre-packaged application on the cluster, first copy it over using SCP:

```
scp -i mykeypair.pem app.jar root@masternode:~
```

Destroying a Cluster

To destroy a cluster launched by `spark-ec2`, run:

```
./spark-ec2 destroy mycluster
```

This will terminate all the instances associated with the cluster (i.e. all instances in its two security groups, `mycluster-master` and `mycluster-slaves`).

Pausing and Restarting Clusters

In addition to outright terminating clusters, `spark-ec2` also lets you stop the Amazon instances running your cluster and then start them again later. Stopping instances shuts them down and makes them lose all data on the "ephemeral" disks, which are configured with an installation of HDFS for `spark-ec2` (see the storage section below). However, the stopped instances retain all data in their root directory (e.g. any files you uploaded there), so you'll be able to quickly return to work.

To stop a cluster, use:

```
./spark-ec2 stop mycluster
```

Then, later, to start it up again:

```
./spark-ec2 -k mykeypair -i mykeypair.pem start mycluster
```

While the Spark EC2 script does not provide commands to resize clusters, you can also resize them by adding or removing machines to the `mycluster-slaves` security group. To add machines, first stop the cluster, then use the AWS management console to right-click one of the slave nodes and select "launch more like this". This will create more instances in the same security group. Then use `spark-ec2 start` to start your cluster. To remove machines, simply terminate them from the AWS console (though beware that this will lose data on the cluster's HDFS installations).

Storage on the Cluster

Spark EC2 clusters come configured with two installations of the Hadoop file system (HDFS) that you can use for scratch space. This can be handy to save data sets in a medium that's faster to access than Amazon S3. The two installations are:

- An "ephemeral" HDFS installation using the ephemeral drives on the nodes. Most Amazon instance types come with a substantial amount of local space attached on "ephemeral" drives that go away if you stop the instance. This installation of HDFS uses this space, giving you a significant amount of scratch space, but it loses all data when you stop and restart the EC2 cluster. It is installed in the `/root/ephemeral-hdfs` directory on the nodes, where you can use the `bin/hdfs` command to access and list files. You can also view the web UI and HDFS URL for it at `http://masternode:50070`.
- A "persistent" HDFS installation on the root volumes of the nodes. This instance persists data even through cluster restarts, but is generally smaller and slower to access than the ephemeral one. It is good for medium sized datasets that you do not wish to download multiple times. It is installed in `/root/persistent-hdfs`, and you can view its web UI and HDFS URL for it at `http://masternode:60070`.

Apart from these, you will most likely be accessing data in Amazon S3, which you can do using the `s3n://` URI scheme in Spark. Refer to the section on Amazon S3 for details.

Which Cluster Manager to Use?

Spark's cluster managers offer a variety of options for deploying applications. If you are starting a new deployment and looking to choose a cluster manager, we recommend the following guidelines:

- Start with a standalone cluster if this is a new deployment. Standalone mode is the easiest to set up and will provide almost all the same features as the other cluster managers if you are only running Spark.

- If you would like to run Spark alongside other applications, or to use richer resource scheduling capabilities (e.g. queues), both YARN and Mesos provide these features. Of these, YARN will likely be preinstalled in many Hadoop distributions.
- One advantage of Mesos over both YARN and standalone mode is its fine-grained sharing option, which lets interactive applications such as the Spark shell scale down their CPU allocation between commands. This makes it attractive in environments where multiple users are running interactive shells.
- In all cases, it is best to run Spark on the same nodes as HDFS for high-speed access to storage. You can install Mesos or the standalone manager on the same nodes manually, or most Hadoop distributions already install YARN and HDFS on the same nodes.

Finally, keep in mind that cluster management is a fast-moving space — by the time this book comes out, there might be more features available for Spark under more of the cluster managers. The methods of submitting applications described here will not change, but you can check the official documentation for your Spark release to see the latest options.

CHAPTER 8
Spark Streaming

This chapter introduces Spark Streaming, Spark's stream processing module allows scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from streaming sources, and processed data can be pushed out to file systems, databases, and live dashboards.

Spark Streaming introduces `DStreams` to represent a stream of data. It is essentially a continuous of series of RDDs, where each RDD represents a time slice of the stream's data.

Similar to `RDDs`, we express our streaming computation through transformations on `DStreams`. Spark Streaming provides a number of standard transformation directly on `DStreams` and allows any arbitrary `RDD` transformations to be applied to the stream's `RDDs`. The concepts we learned in chapters Chapter 3 and Chapter 4 are key to understanding Spark Streaming.

For input, there is built in support for Apache Kafka, Apache Flume, Amazon Kinesis, HDFS, sockets and more. Furthermore, custom receivers can be built for arbitrary sources of streaming data. For output, since our transformed `DStreams` are comprised of RDDs, we can either use any RDD actions covered in Chapter 5, or implement custom logic to push the transformed data to external systems.

As of Spark 1.1.0, Spark Streaming is only available in Java and Scala. Python support is available in Spark 1.2.0 or above.

Simple Example

Spark Streaming, while part of Spark, ships as a separate Maven artifact and has some additional imports you will want to add to your project.

Example 10-1. Maven coordinates for spark-streaming

```
groupId = org.apache.spark
artifactId = spark-streaming_2.10
version = 1.1.0
```

Example 10-2. Scala streaming imports

```
// Import streaming
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
```

Example 10-3. Java streaming imports

```
// Import the Spark Streaming Context
import org.apache.spark.streaming.api.java.JavaStreamingContext;
// Import the different types of DStreams
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
// Import the Duration class
import org.apache.spark.streaming.Duration;
```

Before we dive into all of the cool things we can do with `DStreams`, let us consider a simple example.

First, we will create a `StreamingContext` which is the main entry point for all streaming functionality. This also sets up and underlying `SparkContext` that it will use to process the data. Next, we will use `socketTextStream` to create a DStream based on streaming text data received from a server running at port 7777 of the local machine. Then we will *transform* the DStream with `filter` to get only the lines that contain the word "error". Finally, we will apply the *output operation* `print` to print some of the filtered lines to the console.

Example 10-4. Scala streaming filter example for printing lines containing "error"

```
// Create a StreamingContext with a 1 second batch size
val ssc = new StreamingContext(conf, Seconds(1))
// Create a DStream using data received after connecting to port 7777 on the local machine
val lines = ssc.socketTextStream("localhost", 7777)
// Filter our DStream for lines with "error"
val errorLines = lines.filter(_.contains("error"))
// Print out the lines with errors, which causes this DStream to be evaluated
errorLines.print()
```

Example 10-5. Java streaming filter example for printing lines containing "error"

```
// Create a StreamingContext with a 1 second batch size
JavaStreamingContext jssc = new JavaStreamingContext(conf, new Duration(1000));
// Create a DStream from all the input on port 7777
JavaDStream<String> lines = jssc.socketTextStream("localhost", 7777);
// Filter our DStream for lines with "error"
JavaDStream<String> errorLines = lines.filter(new Function<String, Boolean>() {
    public Boolean call(String line) {
      return line.contains("error");
    }});
// Print out the lines with errors, which causes this DStream to be evaluated
errorLines.print();
```

This only sets up the computaton that will be done when the system receives data. To start receiving and processing data, we will explicitly call `start` on the streaming context. Then, Spark Streaming will start to schedule Spark jobs on the underlying `SparkContext`. This will occur in a separate thread, so to keep our application from exiting we also need to call `awaitTermination` to wait for our streaming computation to finish. To stop the computation, we will need to call `stop`.

Example 10-6. Scala streaming filter example for printing lines containing "error"

```
// Start our streaming context and wait for it to "finish"
ssc.start()
// Wait for 30 seconds then exit. To run forever call without a timeout.
ssc.awaitTermination(30000)
ssc.stop()
```

Example 10-7. Java streaming filter example for printing lines containing "error"

```
// Start our streaming context and wait for it to "finish"
jssc.start();
// Wait for 30 seconds then exit. To run forever call without a timeout.
jssc.awaitTermination(30000);
// Stop the streaming context
jssc.stop();
```

Note that a streaming context can only be started once, and must be started after we set up all the DStreams and output operations.

Now that we have our simple streaming application we can go ahead and run it:

Example 10-8. Linux/Mac Run streaming app and provide data

```
tail -f /var/log/apache/access.log | nc -l 7777 & \
$SPARK_SUBMIT_SCRIPT --class com.oreilly.learningsparkexamples.scala.StreamingLogInput \
$ASSEMBLY_JAR local[4]
```

Windows users can use the ncat command in place of the nc command. ncat is available as part of nmap.

If you don't have Apache access logs, we can get the same effect by running the script `./bin/fakelogs.sh` or `./bin/fakelogs.cmd` provided in our repo or run `nc -lk 7777` in another terminal window and enter input manually.

High Level Architecture

Spark Streaming has a micro-batch architecture, the streaming computation is treated as a continuous series of batch computations on small batches of data. Spark Streaming's `Receivers` receive data from input sources and divide them into small batches. The size of these batches are 500 millieconds or more, and is configurable by These batches are treated as RDDs and processed using Spark jobs. The processed results are pushed out to external systems in batches.

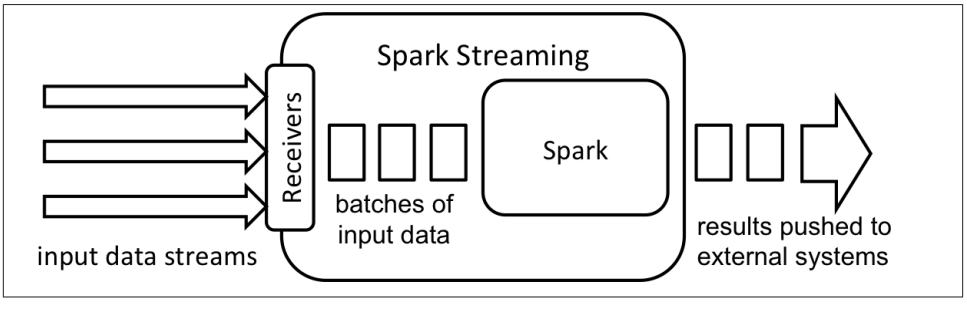

Figure 10-1. High-level architecture of Spark Streaming

As already seen in the previous example, the programming abstraction provided by Spark Streaming is a `DStream` (short of *Discretized Streams*), which is a continuous series of RDDs. DStreams can either be created from input sources (input batches are the RDDs), or by applying `transformations` to existing DStreams. DStreams have many of the transformations that you saw in the Programming with RDDs chapter, and more RDD transformations can be used using `DStream.transform`.

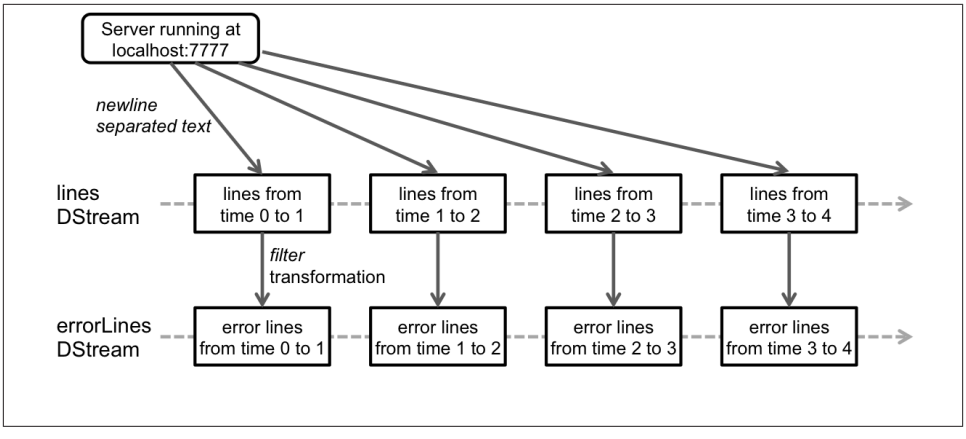

Figure 10-2. Basic Streaming Flow

Additionally, DStreams also have stateful transformations that allow data to be aggregated across batches. These are all explored in detail later in the chapter.

In the previous example, text data is received from the server forms RDDs of the lines DStream. Each RDD is then filtered to form the RDDs of the errorLines DStream. As we run this example, we should see something similar to the following output:

Example 10-9. Log output from running streaming log input example

```
Time: 1413833674000 ms

71.19.157.174 - - [24/Sep/2014:22:26:12 +0000] "GET /error78978 HTTP/1.1" 404 505 "-" "Mozilla/5.0 (X1

....

Time: 1413833675000 ms

71.19.164.174 - - [24/Sep/2014:22:27:10 +0000] "GET /error78978 HTTP/1.1" 404 505 "-" "Mozilla/5.0 (X1

....
```

This output nicely illustrates the micro-batch architecture of Spark Streaming. We can see the filtered logs being printed every second, since we set the batch duration as 1 second when we created the `StreamingContext`. The Spark UI also shows the individual jobs that Spark Streaming runs.

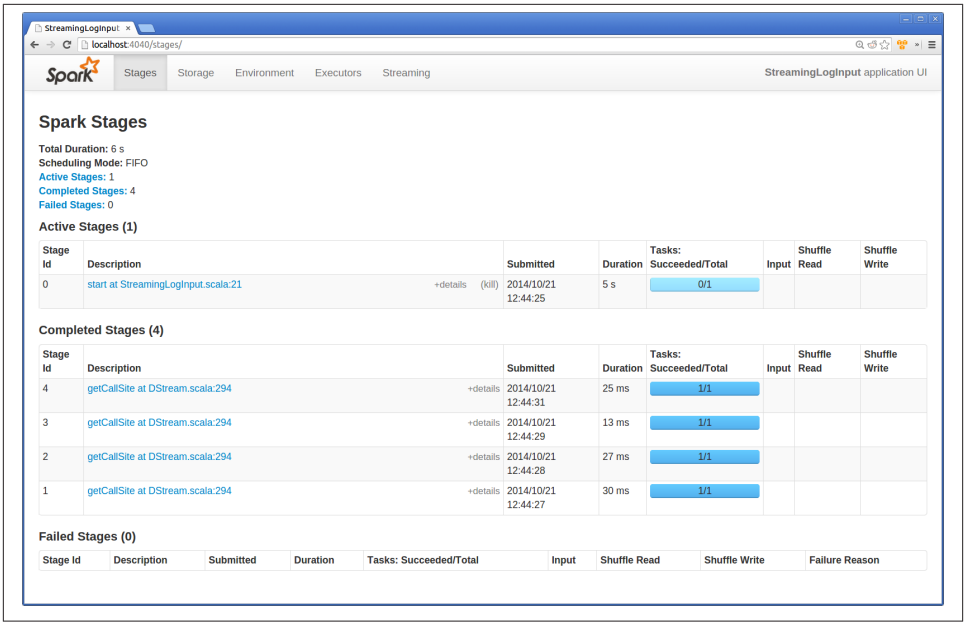

Figure 10-3. Regular Spark UI when running a Streaming Job

Since all DStream transformations translate to RDD transformations, Spark Streaming has the same fault-tolerance properties for DStreams as Spark has for RDDs. By default, all the received data is replicated within the memory of the Spark cluster. This allows the input datasets to be tolerant to single worker failures, and all transformed data to be recoverable by recomputation. The same technique is also used to work around stragglers, so if a node is processing too slowly the work may be assigned to another node.

Next, we will explore the various transformations in detail, discuss the output operations and then elaborate on input sources.

Transformations

Transformations on `DStreams` can be grouped into either stateless or stateful transformations. Stateless transformations are the ones where the processing of each batch does not depend on the data of its previous batches. In contrast, stateful transformations use the data or the intermediate results of previous batches to compute the results of the current batch.

Stateless Transformations

Stateless transformations do not require any additional information to be kept track of by Spark beyond the information in the underlying RDDs themselves. All of the transformations we have done on RDDs are stateless transformations.

Stateless Single DStream Transformations

Table 10-1. Stateless Single DStream transformations

Function name	Purpose	Scala Example	Java Example
map	Apply a function to each element in the DStream and return a DStream of the result	`ds.map(x => x + 1)`	`ds.map(new AddOneFunction())`
flatMap	Apply a function to each element in the DStream and return a DStream of the contents of the iterators returned. Often used for tokenization.	`ds.flatMap(x => x.split(" "))`	`ds.map(new SplitWordsFunc())`
filter	Return an DStream consisting of only elements which pass the condition passed to filter	`ds.filter(x => x != 1)`	`ds.filter(new NotOneFunc())`
repartition	Change the number of partitions of the DStream (either make larger for more parallelism or smaller for less overhead). Covered in "Simple Performance Considerations" on page 153.	`ds.repartition(10)`	`ds.repartition(10)`

Many transformations that we have on `RDDs` are also available directly on `DStreamss` and we have a table of Table 10-1. For processing our log input in the example, we can use `map` on our `DStream`.

Example 10-10. Scala map on DStream

```
val accessLogDStream = logData.map(line => ApacheAccessLog.parseFromLogLine(line))
```

Example 10-11. Java map on DStream

```
JavaDStream<ApacheAccessLog> accessLogsDStream
  = logData.map(new Functions.ParseFromLogLine());
```

Stateless Key-Value DStream Transformations

Additional functions on Key-Value `DStreams` are available. The transformations available are similar to those Chapter 4. As with RDDs, in Java it is necessary to use `to Pair` version of the functions to create a `JavaPairDStream` where as in Scala implicit conversions are used to provide this additional functionality.

Table 10-2. Stateless Key-Value DStreams

Function name	Purpose
`reduceByKey`	Combine values with the same key together
`groupByKey`	Group together values with the same key
`mapValues`	Apply a function to each value of a Pair DStream without changing the key
`flatMapValues`	Apply a function which returns an iterator to each value of a Pair DStream and for each element returned produce a key-value entry with the old key. Often used for tokenization.

Example 10-12. Scala reduceByKey on DStream

```
val ipAddressesDStream = accessLogsDStream.map(entry => (entry.getIpAddress(), 1))
val ipAddressesCountsDStream = ipAddressesDStream.reduceByKey((x, y) => x + y)
```

Example 10-13. Java reduceByKey on DStream

```
public static final class IpTuple implements PairFunction<ApacheAccessLog, String, Long> {
  @Override
  public Tuple2<String, Long> call(ApacheAccessLog log) {
    return new Tuple2<>(log.getIpAddress(), 1L);
  }
}

JavaPairDStream<String, Long> ipAddressesDStream =
  accessLogsDStream.mapToPair(new IpTuple());
JavaPairDStream<String, Long> ipAddressesCountsDStream =
  ipAddressesDStream.reduceByKey(new LongSumReducer());
```

One of the most useful and general stateless operation is `transform` which allows us to apply any arbitrary RDD-to-RDD operation on a DStream. When we call `transform` on a `DStream`, we need to provide a function that takes a RDD as input and returns another RDD as output. This function will be called on each RDD of a DStream, thus creating a new series of result RDDs. This new series will form the DStream returned by `tranform`. For Java, there is also a `transformToPair` function for returning Key-Value `DStreams`. Using this, we take our access log `DStream` and construct a `DStream` of `ipAddresses` and counts.

Example 10-14. Scala applying transform on a DStream

```
val ipAddressesRawDStream = accessLogsDStream.transform { rdd =>
  rdd.map(accessLog => (accessLog.getIpAddress(), 1)).reduceByKey((x, y) => x + y)
}
```

Example 10-15. Java applying transform on a DStream

```
public static final JavaPairRDD<String, Long> ipAddressCount(
    JavaRDD<ApacheAccessLog> accessLogRDD) {
  return accessLogRDD.mapToPair(new IpTuple()).reduceByKey(new LongSumReducer());
}
```

```
...
// A DStream of ipAddressCounts.
JavaPairDStream<String, Long> ipAddressesRawDStream = accessLogsDStream.transformToPair(
  new Function<JavaRDD<ApacheAccessLog>, JavaPairRDD<String, Long>>(){
  public JavaPairRDD<String, Long> call(JavaRDD<ApacheAccessLog> rdd) {
    return ipAddressCount(rdd);
  }});
```

Stateless DStream Joins Transformations

We can combine data from multiple `DStreams` together, as with RDDs. On Key-Value `DStreams`, we have the same join-related transformations as we have on RDDs, namely `coGroup`, `join`, `leftOuterJoin`, and `rightOuterJoin`. Joins are performed between the underlying RDDs of each `DStream` in each batch. The "Joins" on page 58 explains joins on RDDs and the same properties hold on `DStreams`.

Let us consider a simple inner join between two `DStreams`. From our example we have data keyed by IP address, and we can join the request count against the bytes transferred.

Example 10-16. Scala join two DStreams

```
val ipAddressesBytesDStream =
  accessLogsDStream.map(entry => (entry.getIpAddress(), entry.getContentSize()))
val ipAddressesBytesSumDStream =
  ipAddressesBytesDStream.reduceByKey((x, y) => x + y)
val ipBytesRequestCountDStream =
  ipAddressesRawDStream.join(ipAddressesBytesSumDStream)
```

Example 10-17. Java join two DStreams

```
JavaPairDStream<String, Long> ipBytesDStream =
  accessLogsDStream.mapToPair(new IpContentTuple());
JavaPairDStream<String, Long> ipBytesSumDStream =
  ipBytesDStream.reduceByKey(new LongSumReducer());
JavaPairDStream<String, Tuple2<Long, Long>> ipBytesRequestCountDStream =
  ipBytesSumDStream.join(ipAddressesCountsDStream);
```

When combining data between DStreams they must have the same slide duration (discussed in "Windowed Transformations" on page 138) and must be from the same streaming context.

We can also merge the contents of two different `DStreams` of the same type with the `union` operator as in regular Spark. For merging multiple `DStreams`, we have `StreamingContext.union`. Note that similar to RDD unions, applying union on DStream does not move around any data.

Finally, for arbitrary stateless multi-DStream operations, we have `DStream.transformWith` (for two DStreams) and `StreamingContext.transform` (for any number of streams).

Stateful Transformations

Stateful transformations are the ones where some of the data of previous batch is used to generate the results of the current batch. For stateful operations which require we keep track of some explicit bits of state, namely `updateStateByKey` and the optimized `reduceByKeyAndWindow`, we need give Spark Streaming a place to store this data. We do this by enabling checkpointing which gives Spark a place to store this state information. In cases where Spark Streaming can just re-use the previous RDDs, like a simple `window`, we don't have any additional requirements.

Example 10-18. Scala set up checkpointing

```
ssc.checkpoint(opts.CheckpointDirectory)
```

Example 10-19. Java set up checkpointing

```
jssc.checkpoint(Flags.getInstance().getCheckpointDirectory());
```

Windowed Transformations

Windowed operations in Spark are stateful as they need to keep information from the previous time periods around. In our example we use windowed operations to keep track of the most common response codes, content sizes, and clients in a given window.

All windowed operations need two parameters, window duration and sliding duration, both of which must me a multiple of the DStream's batch interval. The window duration controls how many previous batches of data are considered, namely the last `windowDuration / batchInterval`. If we had a source `DStream` with a batch interval of 10 seconds and wanted to create a sliding window of the last 30 seconds (or last 3 batches) we would set the `windowDuration` to 30 seconds. The sliding duration, which defaults to the DStream's batch interval, controls how frequently the new `DStream` operation is performed. If we had the source `DStream` with a batch interval of 10 seconds and wanted to only compute our window on every second batch, we would set our sliding interval to 20 seconds. The window duration and sliding duration respectively control the amount of historic data and the frequency at which the window is computed.

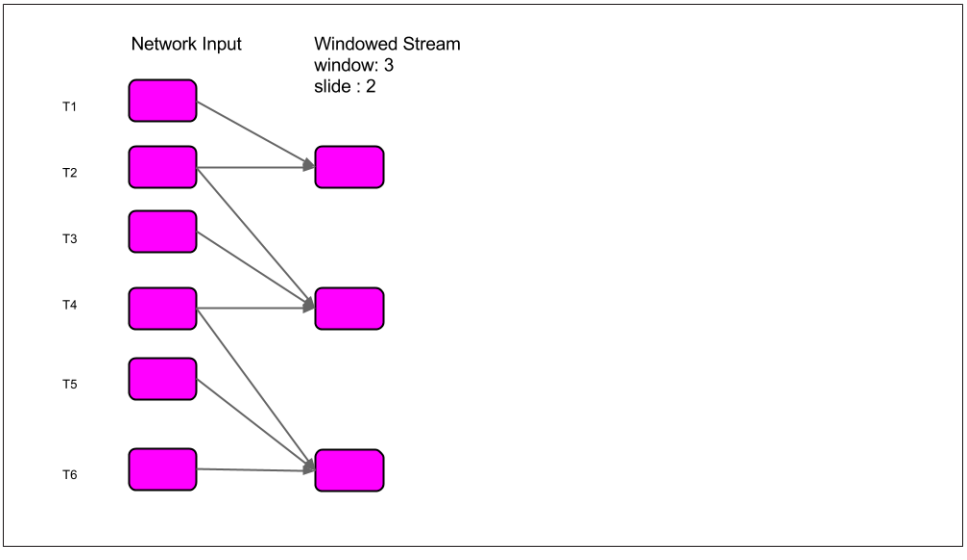

Figure 10-4. Window example image

The simplest window operation we can do on a `DStream` is `window`, which returns a new DStream with the data for the requested window. While we can build all other windowed operations on top of this, Spark Streaming provides a number of other windowed operations for both convenience and efficiency purposes. `window` is often used along with `transform` when we wish to re-use our batch processing code for windowed streaming applications.

Example 10-20. Scala window batch processing re-use example

```
def responseCodeCount(accessLogRDD: RDD[ApacheAccessLog]) = {
  accessLogRDD.map(log => (log.getResponseCode(), 1)).reduceByKey((x, y) => x + y)
}
...

val accessLogsWindow = accessLogsDStream.window(
  opts.getWindowDuration(), opts.getSlideDuration())
val responseCodeCount = accessLogsWindow.transform(rdd => responseCodeCount(rdd))
```

Example 10-21. Java window batch processing re-use example

```
public static final JavaPairRDD<Integer, Long> responseCodeCount(
      JavaRDD<ApacheAccessLog> accessLogRDD) {
  return accessLogRDD
    .mapToPair(new ResponseCodeTuple())
    .reduceByKey(new LongSumReducer());
}
...
JavaDStream<ApacheAccessLog> windowDStream = accessLogsDStream.window(
```

```
    Flags.getInstance().getWindowLength(),
    Flags.getInstance().getSlideInterval());
// use a transform for the response code count
JavaPairDStream<Integer, Long> responseCodeCountTransform = accessLogsDStream.transformToPair(
  new Function<JavaRDD<ApacheAccessLog>, JavaPairRDD<Integer, Long>>() {
    public JavaPairRDD<Integer, Long> call(JavaRDD<ApacheAccessLog> logs) {
          return responseCodeCount(logs);
    }
 });
```

`reduceByWindow` and `reduceByKeyAndWindow` allow us to perform reduce transformations using the window. Traditional reduces apply to all of the data in our window, but we also have a special form which allows us to compute the reduction incrementally, by considering only the new data coming into the window and the data which is going out. This special form is useful when we have an inverse of the our reduction, e.g. - for +, and we have a large window. If there isn't a simple inverse of our reduce function, for example if we kept a set of the words seen, then we can use the regular form which will recompute our reduction on all of the data in our window. In our example, we use these two functions to collect visit counts for each IP address in a window.

Example 10-22. Scala visit counts per IP address

```
val ipAddressesDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress(), 1))
val ipAddressesCountDStream = ipAddressesDStream.reduceByKeyAndWindow(
  {(x, y) => x + y}, // Adding elements in the new batches entering the window
  {(x, y) => x - y}, // Removing elements from the oldest batches exiting the window
  opts.getWindowDuration(), // Window duration
  opts.getSlideDuration() // slide duration
)
ipAddressesCountDStream.print()
```

Example 10-23. Java visit counts per IP address

```
JavaPairDStream<String, Long> ipAddressPairDStream = accessLogsDStream.mapToPair(
  new PairFunction<ApacheAccessLog, String, Long>() {
    public Tuple2<String, Long> call(ApacheAccessLog entry) {
      return new Tuple2(entry.getIpAddress(), 1L);
    }});
JavaPairDStream<String, Long> ipAddressesCountDStream = ipAddressPairDStream.reduceByKeyAndWindow(
  // Adding elements in the new slice
  new Function2<Long, Long, Long>() {
    public Long call(Long v1, Long v2) {
      return v1+v2;
    }},
  // Removing elements from the oldest slice
  new Function2<Long, Long, Long>() {
    public Long call(Long v1, Long v2) {
      return v1-v2;
    }},
  Flags.getInstance().getWindowLength(),
```

```
  Flags.getInstance().getSlideInterval());
ipAddressesCountDStream.print();
```

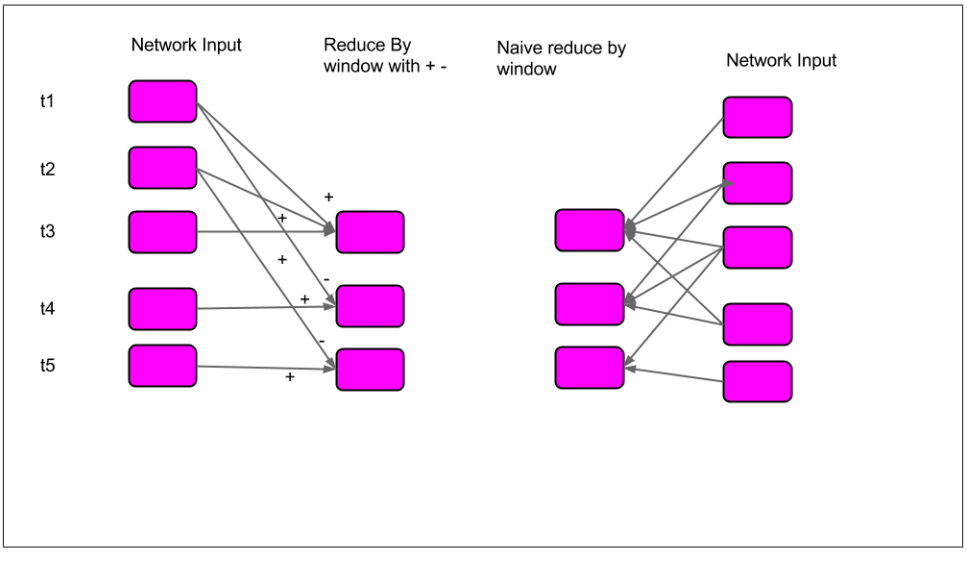

Figure 10-5. Difference between naive reduce by window and optimized reduce by window

If you want to aggregate/reduce over a window, then applying `reduceByKeyAndWindow` is usually more efficient than applying `window(…).reduceByKey`. And if your aggregation can be calculated incrementally (that is, there is a inverse reduce function), then usually `reduceByKeyAndWindow` with inverse reduce function is more efficient than the regular `reduceByKeyAndWindow`.

`countByWindow` and `countByValueAndWindow` are implemented using `reduceByWindow` and `reduceByKeyAndWindow` respectively. `countByWindow` gives us a `DStream` representing the number of elements in the window of the source `DStream`. We will use `countByWindow` to give us the number of requests processed in our window. `countByValueAndWindow` gives us a `DStream` with the number of each value, which we will use to compute the number of requests from each IP address.

Example 10-24. Scala Windowed Count Operations

```
val ipAddressesDStream = accessLogsDStream.map{entry => entry.getIpAddress()}
val ipAddressRequestCount = ipAddressesDStream.countByValueAndWindow(opts.getWindowDuration())
val requestCount = accessLogsDStream.countByWindow()
requestCount.print()
ipAddressesRawDStream.print()
```

Example 10-25. Java Windowed Count Operations

```
JavaDStream<String> ipAddresses = accessLogsDStream.map(
  new Function<ApacheAccessLog, String>() {
    public String call(ApacheAccessLog entry) {
      return entry.getIpAddress();
    }});
JavaDStream<Long> requestCount = accessLogsDStream.countByWindow(
  Flags.getInstance().getWindowLength(), Flags.getInstance().getSlideInterval());
JavaPairDStream<String, Long> ipAddressRequestCount = ipAddresses.countByValueAndWindow(
  Flags.getInstance().getWindowLength(), Flags.getInstance().getSlideInterval());
requestCount.print();
ipAddressRequestCount.print();
```

UpdateStateByKey transformation

`UpdateStateByKey` is the most obvious stateful transformation that allows us to maintain arbitrary per-key state. We have to provide a function that defines how the previous state and the new data is used to generate the new state. Specifically, this function takes as parameter a sequence of values (potentiall empty) and an optional previous state, and has to return an optional updated state. The state for a key will get removed if the state returned is an empty option. The RDDs of the new DStream generated by `updateStateByKey` contains the per-batch snapshots of the state for all the keys.

In our Log processing example we use `updateStateByKey` to keep track the response code counts (success, failure, etc.) through out the entire life of our streaming program.

Example 10-26. Scala keep track of response codes

```
def computeRunningSum(values: Seq[Long], state: Option[Long]) = {
  Some(values.reduce((x, y) => x + y) + state.getOrElse(0L))
}
...
val responseCodeDStream = accessLogsDStream.map(log => (log.getResponseCode(), 1L))
val responseCodeCountDStream = responseCodeDStream.updateStateByKey(computeRunningSum _)
```

Example 10-27. Java keep track of response codes

```
final class ComputeRunningSum implements Function2<List<Long>,
  Optional<Long>, Optional<Long>> {
  @Override
  public Optional<Long> call(List<Long> nums, Optional<Long> current) {
    long sum = current.or(0L);
    for (long i : nums) {
      sum += i;
    }
    return Optional.of(sum);
  }
};
...
```

```
JavaPairDStream<Integer, Long> responseCodeCountDStream = accessLogsDStream.transformToPair(
  new Function<JavaRDD<ApacheAccessLog>, JavaPairRDD<Integer, Long>>() {
    public JavaPairRDD<Integer, Long> call(JavaRDD<ApacheAccessLog> rdd) {
      return Functions.responseCodeCount(rdd);
    }})
  .updateStateByKey(new Functions.ComputeRunningSum());
```

Output Operations

Output operations specify what needs to be done with the final transformed data, either push it out to an external databased, or simply print to screen. Output operations forces all the lazy DStream transformations to be evaluated when the context is started.

If no output operation is applied on a DStream and any of its descendants, then those DStreams will not be evaluated. And if there are no output operations set in a streaming context, then the context will have nothing to process and hence will not start.

A common debugging output operation that we have used already is `print`. This grabs the first 10 elements from each batch of the `DStream` and prints the results.

Once we have our program debugged and we want to save the output. In core Spark, our save operations take a path where the `RDD` is saved. Spark Streaming has similar save operations for `DStreams`, each of which takes a directory and an optional suffix. The results of each batch is saved as files in the directory as different files with the time and the suffix in the file name. For example, we can save our IP address request count information out as text files.

Example 10-28. Scala example to save DStream to text files

```
ipAddressRequestCount.saveAsTextFiles(opts.OutputDirectory + "/ipAddressRequestCountsTXT")
```

Spark Streaming doesn't have the built in `saveAsSequencefile` function, but we can use the `saveAsHadoopFiles` to accomplish the equivalent. This is similar to how we save sequence files in java.

Example 10-29. Scala example to save a sequence file from a DStream

```
val writableIpAddressRequestCount = ipAddressRequestCount.map {case (ip, count) =>
  (new Text(ip), new LongWritable(count)) }
writableIpAddressRequestCount.saveAsHadoopFiles[SequenceFileOutputFormat[Text, LongWritable]](
  opts.OutputDirectory + "/ipAddressRequestCounts", "pandas")
```

Example 10-30. Java extample to save a sequence file from a DStream

```
JavaPairDStream<Text, LongWritable> writableDStream = ipAddressesDStream.mapToPair(
  new PairFunction<Tuple2<String, Long>, Text, LongWritable>() {
```

```
    public Tuple2<Text, LongWritable> call(Tuple2<String, Long> e) {
      return new Tuple2(new Text(e._1()), new LongWritable(e._2()));
    }
  });
class OutFormat extends SequenceFileOutputFormat<Text, LongWritable> {
};
writableDStream.saveAsHadoopFiles(outDir, "pandas",
                                  Text.class, LongWritable.class,
                                  OutFormat.class);
```

`foreachRDD` is the generic output operation that allows us to arbitrary computations on the RDDs on the DStream. It is similar to `transform` but it does not return a new DStream. It is a common pattern in Spark Streaming since it allows us to reuse `actions` from regular Spark, mostly focused around saving the processed data. `foreachRDD` optionally gives us the time of the current batch, allowing us to output each time period to a different directory. saveAsTextFiles+ and the other native `DStream` saving functions are implemented using `foreachRDD` . For any custom external system, we can use `foreachRDD` to directly push that data to those systems using the following pattern.

Example 10-31. Scala design pattern to efficiently save data to external system

```
ipAddressRequestCount.foreachRDD { rdd =>
  rdd.foreachPartition { partition =>
    // Open connection to system
    partition.foreach { iteram =>
      // Use connection to insert item
    }
    // Close connection
  }
}
```

Example 10-32. Java example to save a sequence file from a DStream

```
JavaPairDStream<Text, LongWritable> writableDStream = ipAddressesDStream.mapToPair(
  new PairFunction<Tuple2<String, Long>, Text, LongWritable>() {
    public Tuple2<Text, LongWritable> call(Tuple2<String, Long> e) {
      return new Tuple2(new Text(e._1()), new LongWritable(e._2()));
    }
  });
class OutFormat extends SequenceFileOutputFormat<Text, LongWritable> {
};
writableDStream.saveAsHadoopFiles(outDir, "pandas",
                                  Text.class, LongWritable.class,
                                  OutFormat.class);
```

Output operations of DStreams are similar to actions of RDDs in the sense that they force the lazy evaluation of the RDDs/DStreams. However, there is subtle difference. RDD's actions are blocking methods that force Spark jobs to be launched immediately. Output operations only *sets up* the DStreams to be evaluated with Spark jobs, and the jobs are actually launched only after the streaming context has been started.

Input Sources

Spark Streaming has built in support for a number of different data sources. However, these sources are divided into two categories.

- *Core sources* - Sources whose functionalities are available through the Spark Streaming Maven artifact
- *Additional sources* - Sources whose functionalities are available only through additional Maven artifacts.

This section walks through some of these sources.

Core Sources

The methods to create DStream from the core sources are all available on the `StreamingContext`. We have already expored one of these sources in the example - sockets. Here we discuss two more - HDFS files and Akka Actors.

Stream of HDFS files

Since Spark supports reading from any HDFS-compatible file system, Spark Streaming naturally allows a stream to be created from files written in a directory of a HDFS-compatible file system. This is a popular option due its support of a wide variety of backends, especially for log data which we would copy to HDFS anyways. For Spark Streaming to work with the data, it needs to have a consistent date format for the directory names and the files have to be created atomically, e.g. by moving the file,[1] in the directory Spark is monitoring. We can change our previous example that was reading from a network socket to handle new log files as they show up in a directory instead.

Example 10-33. Scala example of streaming text files written to a directory

```
val logData = ssc.textFileStream(logDirectory)
```

1. Atomically means that the entire operation happens at once. This is important here since if Spark Streaming were to start processing the file and then more data were to appear Spark Streaming wouldn't notice the additional data.

Example 10-34. Java example of streaming text files written to a directory

```
JavaDStream<String> logData = jssc.textFileStream(logsDirectory);
```

We can use the provided `./bin/fakelogs_directory.sh` script to fake the logs, or if we have real log data we could replace the rotator with an `mv` command to rotate the logs files into the directory we are monitoring.

In addition to text data, in Scala we can also read any Hadoop Input format. As with "Hadoop Input and Output Formats" on page 81 we simply need to provide Spark Streaming with the `Key`, `Value`, and `InputFormat` classes. If, for example, we had a previous streaming job process the logs and save the bytes transferred with the time as the key we could read the data back in like so.

Example 10-35. Scala example of streaming sequence files written to a directory

```
ssc.fileStream[LongWritable, IntWritable,
   SequenceFileInputFormat[LongWritable, IntWritable]](inputDirectory).map {
   case (x, y) => (x.get(), y.get())
}
```

Java support for other Hadoop Input formats is expected in Spark 1.2

Akka Actor Stream

The second core receiver is `actorStream` and it allows use Akka actor's as a source for streaming. To construct an actor stream we create an Akka actor and implement the `org.apache.spark.streaming.receiver.ActorHelper` trait/interface. To copy the input from our actor into Spark Streaming we need to call the `store` function in our actor when we receive new data. Akka Actor streams are less common so we won't go into detail, but you can look at the streaming documentation and the ActorWordCount example in `spark/examples/src/main/scala/org/apache/spark/examples/streaming/ActorWordCount.scala`.

Additional Sources

In addition to the core sources, additional receivers for well-known data ingestion systems are packaged as separate components of Spark Streaming. These receivers are still part of Spark, but require additional packages to be included in your build file. Spark Streaming's current additional receivers include Twitter, Apache Kafka, Amazon Kinesis, Apache Flume, and ZeroMQ. We can include these additional receivers by adding

the Maven artifact `spark-streaming-[projectname]_2.10` with the same version number as Spark.

Apache Kafka

Apache Kafka is popular input source due to its speed and resilience. Using the native support for Kafka, we can easily process the messages for many topics. To use it, we have to include the Maven artifact `spark-streaming-kafka_2.10` to our project. The provided KafkaUtils object works on `StreamingContext` and `JavaStreamingContext` to create a `DStream` of your Kafka messages. Since it can subscribe to multiple topics, the `DStream` it creates consists of pairs of topic and message. To create a stream, we will call the `createStream` method with our streaming context, a string containing comma separated ZooKeeper hosts, the name of our consumer group (a unique name), and a map of topics to number of receiver threads to use for that topic.

Example 10-36. Scala Apache Kafka subscribe to Panda's topic example

```
import org.apache.spark.streaming.kafka._
...
// Create a map of topics to number of receiver threads to use
val topics = List(("pandas", 1), ("logs", 1)).toMap
val topicLines = KafkaUtils.createStream(ssc, zkQuorum, group, topics)
StreamingLogInput.processLines(topicLines.map(_._2))
```

Example 10-37. Java Apache Kafka subscribe to Panda's topic example

```
import org.apache.spark.streaming.kafka.*;
...
// Create a map of topics to number of receiver threads to use
Map<String, Integer> topics = new HashMap<String, Integer>();
topics.put("pandas", 1);
topics.put("logs", 1);
JavaPairDStream<String, String> input = KafkaUtils.createStream(jssc, zkQuorum, group, topics);
input.print();
```

Apache Flume

Spark has two different receivers for use with Apache Flume. They are as follows:

- Push-based receiver - The receiver act as an Avro sink that Flume pushes data to.
- Pull-based receiver - The receiver can pull data from an intermediate custom sink, to which

Both approaches require re-configuring Flume and running the receiver on a node on a configured port (not your existing Spark or Flume ports). To use either of them, we have to include the Maven artifact `spark-streaming-flume_2.10` to our project.

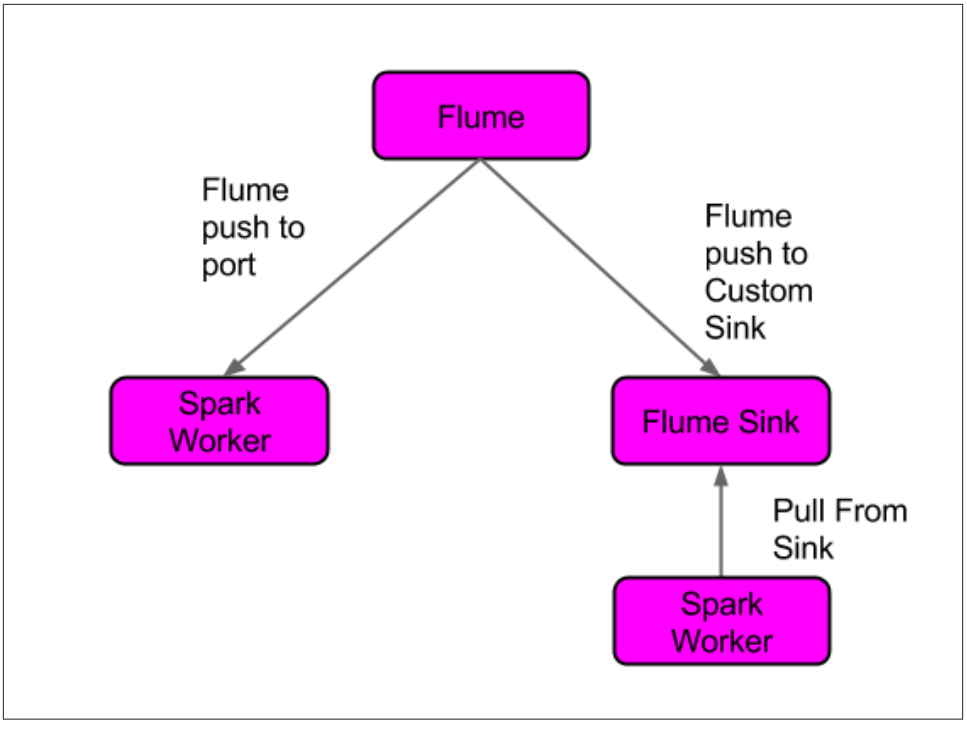

Figure 10-6. Flume receiver options

Push-based receiver

The push-based approach can be set up quickly but does not use transactions to receive data. In this approach, the receiver acts as an Avro sink, and we need to configure Flume to send the data to the Avro sink. The provided `FlumeUtils` object sets up the receiver to be startd on a specific worker's hostname and port. These must match those in our Flume configuration.

Example 10-38. Flume Configuration for Avro sink

```
a1.sinks = avroSink
a1.sinks.avroSink.type = avro
a1.sinks.avroSink.channel = memoryChannel
a1.sinks.avroSink.hostname = receiver-hostname
a1.sinks.avroSink.port = port-used-for-avro-sink-not-spark-port
```

Example 10-39. Scala FlumeUtils agent example

```
val events = FlumeUtils.createStream(ssc, receiverHostname, receiverPort)
```

Example 10-40. Java FlumeUtils agent example

```
JavaDStream<SparkFlumeEvent> events = FlumeUtils.createStream(ssc, receiverHostname, receiverPort)
```

Despite its simplicity, the disadvantage of this approach is its lack of transactions. This increases the chance of losing small amounts of data in case of the failure of the worker node running the receiver. Furthermore, if the worker running the receiver fails, the system will try to launch the receiver at a different location, and Flume will need to be reconfigured to send to the new worker. This is often challenging to set up.

Pull-based receiver

TThe newer pull-based approach (introduced in Spark 1.1) is to set up a specialized Flume sink Spark Streaming will read from, and the receiver will pull the data from the sink. This approach is preferred for resiliency, as the data remains buffered in the sink until Spark Streaming reads and replicates the data and acknowledges the sink via transactions.

To get started, we will need to set up the custom sink as a third party plugin for Flume. The latest directions on installing plugins are in the Flume documentation. Since the plugin is written in Scala we need to add both the plugin and the Scala library to Flume's plugins. For Spark 1.1, the Maven co-ordinates are:

Example 10-41. Maven coordinates for Flume Sink

```
groupId = org.apache.spark
artifactId = spark-streaming-flume-sink_2.10
version = 1.1.0

groupId = org.scala-lang
artifactId = scala-library
version = 2.10.4
```

Once you have the custom flume sink added to a node, we need to configure Flume to push to the sink.

Example 10-42. Flume Configuration for custom sink

```
a1.sinks = spark
a1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
a1.sinks.spark.hostname = receiver-hostname
a1.sinks.spark.port = port-used-for-sync-not-spark-port
a1.sinks.spark.channel = memoryChannel
```

With the data being buffered in the sink we can now use `FlumeUtils` to pull from the sink.

Example 10-43. Scala FlumeUtils custom sink

```
val events = FlumeUtils.createPollingStream(ssc, receiverHostname, receiverPort)
```

Example 10-44. Java FlumeUtils custom sink

```
JavaDStream<SparkFlumeEvent> events = FlumeUtils.createPollingStream(ssc,
  receiverHostname, receiverPort)
```

In either case, the DStream is comprised of https://spark.apache.org/docs/latest/api/java/org/apache/spark/streaming/flume/SparkFlumeEvent.htmlSparkFlumeEvent+s. We can access the underlying +AvroFlumeEvent through event. If our event body was UTF-8 strings we could get the contents as follows:

Example 10-45. Scala SparkFlumeEvent example

```
// Assuming that our flume events are UTF-8 log lines
 val lines = events.map{e => new String(e.event.getBody().array(), "UTF-8")}
```

Custom input sources

In addition to the provided sources, you can also implement your own receiver which is described in the Streaming Custom Receivers guide.

Mulitple sources

As covered in "Stateless DStream Joins Transformations" on page 137 we can combine multiple DStreams. Provided that the input sources are created on the same streaming context, we can have multiple input sources and join or union the resulting DStreams.

Receiver Architecture

It is important to understand how the receivers are executed in the Spark cluster. Each receiver runs as a long running task within Spark's executors, and hence occupies CPU cores allocated to the application. And there needs to be more available cores for processing the data. Resources need to be allocated accordingly. For example, if we want to run 10 receivers in our streaming application, then we have to allocate at least 11 cores. Otherwise, either receiver will not run, and/or the received data will not be processed.

Do not run Spark Streaming programs locally with master configured as "local". This allocates only one CPU and if a receiver is running on it, there is no resource left to process the received data.

Fault Tolerance

There are three different types of failures to consider with Spark Streaming. There is failure of a worker node, and failure of the driver node.

Failure of a worker node

For failure of a worker node, Spark Streaming uses the same techniques as Spark for its fault tolerance. All the data received through Receivers (that is, data from Kafka, Flume, etc.) is replicated among the Spark workers. All RDDs created through transformations

of this replicate input data are tolerant to failure of a worker node, as the RDD lineage allows the system to recompute the lost data all the way from the surviving replica of the input data. Note that for file streams, the files are already in a fault-tolerant file system (like HDFS and S3) and therefore do not need replication.

The fault tolerance of the workers running the receivers is another important consideration. In such a failure, Spark Streaming restarts the failed receivers on other nodes in the cluster. However, whether it looses any of the received data depends on the nature of the source (whether the source can resend data or not) and the implementation of the receiver (whether it updates the source about received data or not). For example, with Flume, one of the main differences between the two receivers is the data loss guarantees. With the receiver-pull-from-sink model, Spark only removes the elements once they have been replicated inside Spark. For the push-to-receiver model, if the receiver fails before the data is replicated some data can be lost. In general, for any receiver, the fault-tolerance properties of the upstream source (transactional, or not) also needs to be considered for ensuring zero data loss.

Failure of the driver node

Failure of the driver node requires we use a different way of creating our Streaming Context. If checkpointing is enabled, which is required for all stateful operations, we can recover our previous context from the checkpointing directory. Instead of simply calling `new StreamingContext` we need to use the `StreamingContext.getOrCreate` function. From our initial example we would change our code as follows:

Example 10-46. Scala Streaming Driver Recovery

```
def createStreamingContext() = {
...
  val sc = new SparkContext(conf)
  // Create a StreamingContext with a 1 second batch size
  val ssc = new StreamingContext(sc, Seconds(1))
}
...
val ssc = StreamingContext.getOrCreate(checkPointdir, createStreamingContext _)
```

Example 10-47. Java Streaming Driver Recovery

```
JavaStreamingContextFactory jscf = new JavaStreamingContextFactory() {
  @Override public JavaStreamingcontext call() {
    ...
    JavaSparkContext sc = new JavaSparkContext(conf);
    // Create a StreamingContext with a 1 second batch size
    JavaStreamingContext jssc = new JavaStreamingContext(sc, new Duration(1000));
    return jssc;
  }};
JavaStreamingContext jssc = JavaStreamingContext.getOrCreate(checkPointdir, jscf);
```

When this code is run the first time, assuming that the checkpoint directory does not exist, the `StreamingContext` will be create by calling the factory function (`createStreamingContext` for Scala, and `jscf` for Java). In the factory, you have to set the checkpoint directory. After the driver fails, the driver needs to be restarted. After restart, this code will find that the checkpoint directory exists, recreate the `StreamingContext` and resume processing.

Note that as of Spark 1.1, the data that was received through Receivers (like Kafka, and Flume) but not processed gets lost when the driver fails. With file sources like HDFS and S3, on recovery, input is automatically recovered from where the driver left off before node loss. Improved support for driver fault-tolerance will be introduced in Spark 1.2.

Its important to note that the driver program is currently not automatically re-launched by default. Either a separate job server or monitoring system, such as Ooyala's Spark Job Server, or launch in supervise mode. To allow Spark to re-launch our driver program on failure we need run in cluster mode instead of client mode, so that our driver is executing on a node in the cluster instead of our local machine.

Example 10-48. Launch in supervise mode

```
./bin/spark-submit --deploy-mode cluster --supervise ...
```

Semantic guarantees

Due to Spark Streaming's worker fault-tolerance guarantees, it can provide exactly-once semantics for all transformations - even if a worker fails and some data get reprocessed, the final transformed result (that is, the transformed RDDs) will be such that the data was processed exactly once.

However, when the transformed result is to be pushed to external systems using output operations, the task pushing the result may get executed multiple times due to failures, and some data can get pushed multiple times. Since this involves external systems, it is beyond the scope of the guarantees that Spark Streaming can provide. We can get around this either by using transactions to atomically push to external systems (that is, atomically push one RDD partition at a time), or by modeling the updates as idempotent operations (such that multiple updates do not affect the outcome).

Streaming UI

The Spark Streaming UI lets us dig into our application and quickly. As referenced in Figure 10-3 the Spark UI shows us the individual jobs that are scheduled by Spark Streaming. When we are running a streaming job there is an additional tab on the application UI labeled `Streaming`. The UI of a streaming twitter to elastic search indexing job is included Figure 10-7.

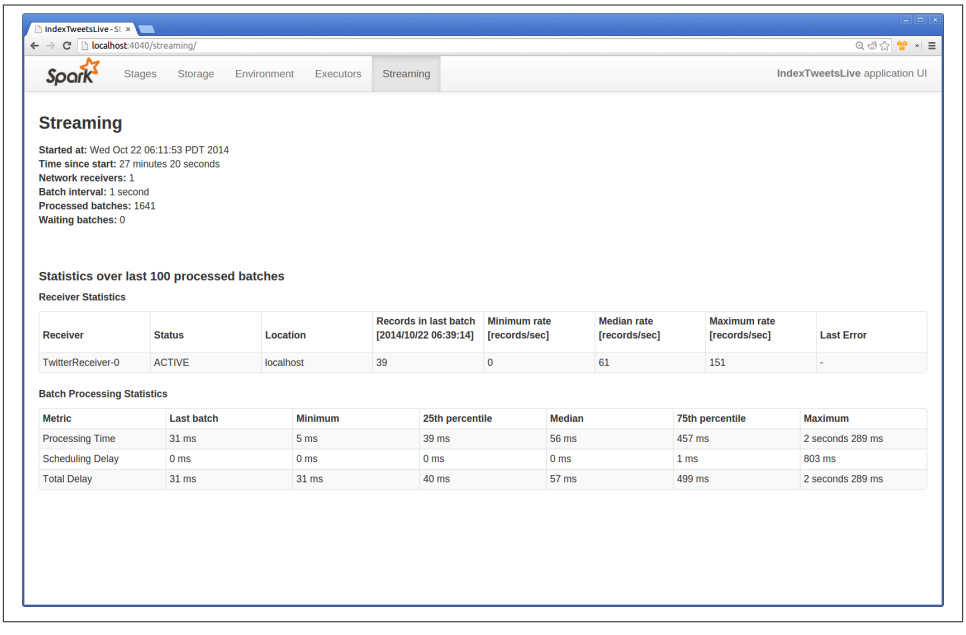

Figure 10-7. Spark streaming UI

The Spark Streaming UI exposes statistics for our batch processing and our receivers. In our example we have one network receiver, and we can see the message process rates. If we were to be falling behind we can see how many records each receiver is able to process. We can also see if a receiver has failed. The batch processing statistics shows us how long our batches take and also breaks out the delay in scheduling the job. If a cluster experiences contention then the scheduling delay may increase.

Simple Performance Considerations

In addition to the existing performance considerations we have discussed in general Spark, Spark Streaming applications have a few specialized tuning options. The most obvious options are related to batch sizing and window sizing, which have a large impact on Spark Streaming performance. A common question is what the minimum batch size can be for Spark Streaming. In general, half a second has proven to be a good minimum batch size for many streaming applications. The best approach is to start with a larger batch size (around 10 seconds) and work your way down to a smaller batch size. If the processing times reported in the streaming ui remain consistent then you can continue to decrease the batch size, but if they are increasing you may have reached the limit for your application.

The most obvious way to reduce the processing time of batches is to increase the parallelism. There are three ways to increase the parallelism

- Increasing the number receivers - Receivers can also sometimes act as a bottleneck if there are too many records for a single machine to read in and distribute. In that case, more receivers can be added by creating multiple input DStreams (which creates multiple receivers), and then applying `union` to merge them into a single stream.
- Explicit repartitioning received data - If receivers cannot be increased further, the received data can be further distributed by explicitly repartitioning the input stream (or the unioned stream if there are multiple input streams) using `DStream.repartition`.
- Increasing parallelism in aggregation - For aggregation operations like `reduceByKey`, the parallelism can be specified, as already discussed for Spark.

Another aspect that can cause problems is Java's Garbage Collection. Unpredictably large pauses due to GC can be minimized by enabling Java's Concurrent Mark-Sweep garbage collector. Furthermore, caching RDDs in serialized form also reduces GC pauses, which is why, by default, RDDs generated by Spark are stored in serialized form.