Frank Kane's Taming Big Data with Apache Spark and Python

Contents

Chapter 1. Getting Started with Spark

Spark is one of the hottest technologies in big data analysis right now, and with good reason. If you work for, or you hope to work for, a company that has massive amounts of data to analyze, Spark offers a very fast and very easy way to analyze that data across an entire cluster of computers and spread that processing out. This is a very valuable skill to have right now.

My approach in this book is to start with some simple examples and work our way up to more complex ones. We'll have some fun along the way too. We will use movie ratings data and play around with similar movies and movie recommendations. I also found a social network of superheroes, if you can believe it; we can use this data to do things such as figure out who's the most popular superhero in the fictional superhero universe. Have you heard of the Kevin Bacon number, where everyone in Hollywood is supposedly connected to a Kevin Bacon to a certain extent? We can do the same thing with our superhero data and figure out the degrees of separation between any two superheroes in their fictional universe too. So, we'll have some fun along the way and use some real examples here and turn them into Spark problems. Using Apache Spark is easier than you might think and, with all the exercises and activities in this book, you'll get plenty of practice as we go along. I'll guide you through every line of code and every concept you need along the way. So let's get started and learn Apache Spark.

Getting set up - installing Python, a JDK, and Spark and its dependencies

Let's get you started. There is a lot of software we need to set up. Running Spark on Windows involves a lot of moving pieces, so make sure you follow along carefully, or else you'll have some trouble. I'll try to walk you through it as easily as I can. Now, this chapter is written for Windows users. This doesn't mean that you're out of luck if you're on Mac or Linux though. If you open up the download package for the book or go to this URL, http://media.sundog-soft.com/spark-python-install.pdf, you will find written instructions on getting everything set up on Windows, macOS, and Linux. So, again, you can read through the chapter here for Windows users, and I will call out things that are specific to Windows, so you'll find it useful in other platforms as well; however, either refer to that `spark-python-install.pdf` file or just follow the instructions here on Windows and let's dive in and get it done.

Installing Enthought Canopy

This book uses Python as its programming language, so the first thing you need is a Python development environment installed on your PC. If you don't have one already, just open up a web browser and head on to https://www.enthought.com/, and we'll install Enthought Canopy:

Enthought Canopy is just my development environment of choice; if you have a different one already that's probably okay. As long as it's Python 3 or a newer environment, you should be covered, but if you need to install a new Python environment or you just want to minimize confusion, I'd recommend that you install Canopy. So, head up to the big friendly download **`Canopy`** button here and select your operating system and architecture:

Â Â Â Â

For me, the operating system is going to be Windows (64-bit). Make sure you choose Python 3.5 or a newer version of the package. I can't guarantee the scripts in this book will work with Python 2.7; they are built for Python 3, so select Python 3.5 for your OS and download the installer:

There's nothing special about it; it's just your standard Windows Installer, or whatever platform you're on. We'll just accept the defaults, go through it, and allow it to become our default Python environment. Then, when we launch it for the first time, it will spend a couple of minutes setting itself up and all the Python packages that we need. You might want to read the license agreement before you accept it; that's up to you. We'll go ahead, start the installation, and let it run.

Once Canopy installer has finished installing, we should have a nice little Enthought Canopy icon sitting on our desktop. Now, if you're on Windows, I want you to right-click on the Enthought Canopy icon, go to **`Properties`** and then to **`Compatibility`** (this is on Windows 10), and make sure **`Run this program as an administrator`** is checked:

This will make sure that we have all the permissions we need to run our scripts successfully. You can now double-click on the file to open it up:

The next thing we need is a Java Development Kit because Spark runs on top of Scala and Scala runs on top of the Java Runtime environment.

Installing the Java Development Kit

For installing the Java Development Kit, go back to the browser, open a new tab, and just search for `jdk` (short for Java Development Kit). This will bring you to the Oracle site, from where you can download Java:

On the Oracle website, click on **`JDK DOWNLOAD`**. Now, click on **`Accept License Agreement`** and then you can select the download option for your operating system:

For me, that's going to be Windows 64-bit and a wait for 198 MB of goodness to download:

Once the download is finished, we can't just accept the default settings in the installer on Windows here. So, this is a Windows-specific workaround, but as of the writing of this book, the current version of Spark is 2.1.1. It turns out there's an issue with Spark 2.1.1 with Java on Windows. The issue is that if you've installed Java to a path that has a space in it, it doesn't work, so we need to make sure that Java is installed to a path that does not have a space in it. This means that you can't skip this step even if you have Java installed already, so let me show you how to do that. On the installer, click on **`Next`**, and you will see, as in the following screen, that it wants to install by default to the `C:\Program Files\Java\jdk` path, whatever the version is:

The space in the `Program Files` path is going to cause trouble, so let's click on the **`Change...`** button and install to `c:\jdk`, a nice simple path, easy to remember, and with no spaces in it:

Now, it also wants to install the Java Runtime environment; so, just to be safe, I'm also going to install that to a path with no spaces.

At the second step of the JDK installation, we should have this showing on our screen:

I will change that destination folder as well, and we will make a new folder called `C:\jre` for that:

Alright; successfully installed. Woohoo!

Now, you'll need to remember the path that we installed the JDK into, which, in our case was `C:\jdk`. We still have a few more steps to go here. So far, we've installed Python and Java, and next we need to install Spark itself.

Installing Spark

Let's us get back to a new browser tab here; head to spark.apache.org, and click on the **`Download Spark`** button:

Now, we have used Spark 2.1.1 in this book. So, you know, if given the choice, anything beyond 2.0 should work just fine, but that's where we are today.

Make sure you get a pre-built version, and select a **`Direct Download`** option so all these defaults are perfectly fine. Go ahead and click on the link next to instruction number **`4`** to download that package.

Now, it downloads a **TGZ** (**Tar in GZip**) file, so, again, Windows is kind of an afterthought with Spark quite honestly because on Windows, you're not going to have a built-in utility for actually decompressing TGZ files. This means that you might need to install one, if you don't have one already. The one I use is called WinRAR, and you can pick that up from www.rarlab.com. Go to the **`Downloads`** page if you need it, and download the installer for WinRAR 32-bit or 64-bit, depending on your operating system. Install WinRAR as normal, and that will allow you to actually decompress TGZ files on Windows:

So, let's go ahead and decompress the TGZ files. I'm going to open up my `Downloads` folder to find the Spark archive that we downloaded, and let's go ahead and right-click on that archive and extract it to a folder of my choosing; just going to put it in my `Downloads` folder for now. Again, WinRAR is doing this for me at this point:

So I should now have a folder in my `Downloads` folder associated with that package. Let's open that up and there is Spark itself. So, you need to install that in some place where you will remember it:

You don't want to leave it in your `Downloads` folder obviously, so let's go ahead and open up a new file explorer window here. I go to my `C` drive and create a new folder, and let's just call it `spark`. So, my Spark installation is going to live in `C:\spark`. Again, nice and easy to remember. Open that folder. Now, I go back to my downloaded `spark` folder and use ***Ctrl*** + ***A*** to select everything in the Spark distribution, ***Ctrl*** + ***C*** to copy it, and then go back to `C:\spark`, where I want to put it, and ***Ctrl*** + ***V*** to paste it in:

Remembering to paste the contents of the `spark` folder, not the `spark` folder itself is very important. So what I should have now is my `C` drive with a `spark` folder that contains all of the files and folders from the Spark distribution.

Well, there are yet a few things we need to configure. So while we're in `C:\spark` let's open up the `conf` folder, and in order to make sure that we don't get spammed to death by log messages, we're going to change the logging level setting here. So to do that, right-click on the `log4j.properties.template` file and select **`Rename`**:

Delete the `.template` part of the filename to make it an actual `log4j.properties` file. Spark will use this to configure its logging:

Now, open this file in a text editor of some sort. On Windows, you might need to right-click there and select **`Open with`** and then **`WordPad`**:

In the file, locate `log4j.rootCategory=INFO`. Let's change this to `log4j.rootCategory=ERROR` and this will just remove the clutter of all the log spam that gets printed out when we run stuff. Save the file, and exit your editor.

So far, we installed Python, Java, and Spark. Now the next thing we need to do is to install something that will trick your PC into thinking that Hadoop exists, and again this step is only necessary on Windows. So, you can skip this step if you're on Mac or Linux.

Let's go to http://media.sundog-soft.com/winutils.exe. Downloading `winutils.exe` will give you a copy of a little snippet of an executable, which can be used to trick Spark into thinking that you actually have Hadoop:

Now, since we're going to be running our scripts locally on our desktop, it's not a big deal, and we don't need to have Hadoop installed for real. This just gets around another quirk of running Spark on Windows. So, now that we have that, let's find it in the `Downloads` folder, clickÂ ***Ctrl*** + ***C*** to copy it, and let's go to our `C` drive and create a place for it to live:

So, I create a new folder again, and we will call it `winutils`:

Now let's open this `winutils` folder and create a `bin` folder in it:

Now in this `bin` folder, I want you to paste the `winutils.exe` file we downloaded. So you should have `C:\winutils\bin` and then `winutils.exe`:

This next step is only required on some systems, but just to be safe, open Command Prompt on Windows. You can do that by going to your Start menu and going down to **`Windows System`**, and then clicking on **`Command Prompt`**. Here, I want you to type `cd c:\winutils\bin`, which is where we stuck our `winutils.exe` file. Now if you type `dir`, you should see that file there. Now type `winutils.exe chmod 777 \tmp\hive`. This just makes sure that all the file permissions you need to actually run Spark successfully are in place without any errors. You can close Command Prompt now that you're done with that step. Wow, we're almost done, believe it or not.

Now we need to set some environment variables for things to work. I'll show you how to do that on Windows. On Windows 10, you'll need to open up the Start menu and go to **Windows System** | **Control Panel** to open up **Control Panel**:

In Control Panel, click on **System and Security**:

Then, click on **System**:

Then click on **Advanced system settings** from the list on the left-hand side:

From here, click on **Environment Variables...**:

We will get these options:

Now, this is a very Windows-specific way of setting environment variables. On other operating systems, you'll use different processes, so you'll have to look at how to install Spark on them. Here, we're going to set up some new user variables. Click on the **New...** button for a new user variable and call it `SPARK_HOME`, as shown as follows, all uppercase. This is going to point to where we installed Spark, which for us is `c:\spark`, so type that in as the **Variable value** and click on **OK**:

We also need to set up `JAVA_HOME`, so click on **`New...`** again and type in `JAVA_HOME` as **`Variable name`**. We need to point that to where we installed Java, which for us is `c:\jdk`:

We also need to set up `HADOOP_HOME`, and that's where we installed the `winutils` package, so we'll point that to `c:\winutils`:

So far, so good. The last thing we need to do is to modify our path. You should have a **`PATH`** environment variable here:

Click on the **`PATH`** environment variable, then on **`Edit...`**, and add a new path. This is going to be `%SPARK_HOME%\bin`, and I'm going to add another one, `%JAVA_HOME%\bin`:

Basically, this makes all the binary executables of Spark available to Windows, wherever you're running it from. Click on **`OK`** on this menu and on the previous two menus. We finally haveÂ everything set up. So, let's go ahead and try it all out in our next step.

Running Spark code

Let's go ahead and start up Enthought Canopy. Once you get to the Welcome screen, go to the **`Tools`** menu and then to **`Canopy Command Prompt`**. This will give you a little Command Prompt you can use; it has all the right permissions and environment variables you need to actually run Python.

So type in `cd c:\spark`, as shown here, which is where we installed Spark in our previous steps:

We'll make sure that we have Spark in there, so you should see all the contents of the Spark distribution pre-built. Let's look at what's in here by typing `dir` and hitting ***Enter***:

Now, depending on the distribution that you downloaded, there might be a `README.md` file or a `CHANGES.txt` file, so pick one or the other; whatever you see there, that's what we're going to use.

We will set up a little simple Spark program here that just counts the number of lines in that file, so let's type in `pyspark` to kick off the Python version of the Spark interpreter:

If everything is set up properly, you should see something like this:

If you're not seeing this and you're seeing some weird Windows error about not being able to find `pyspark`, go back and double-check all those environment variables. The odds are that there's something wrong with your path or with your `SPARK_HOME` environment variables. Sometimes you need to log out of Windows and log back in, in order to get environment variable changes to get picked up by the system; so, if all else fails, try this. Also, if you got cute and installed things to a different path than I recommended in the setup sections, make sure that your environment variables reflect those changes. If you put it in a folder that has spaces in the name, that can cause problems as well. You might run into trouble if your path is too long or if you have too much stuff in your path, so have a look at that if you're encountering problems at this stage. Another possibility is that you're running on a managed PC that doesn't actually allow you to change environment variables, so you might have thought you did it, but there might be some administrative policy preventing you from doing so. If so, try running the set up steps again under a new account that's an administrator if possible. However, assuming you've gotten this far, let's have some fun.

Let's write some Spark code, shall we? We should get some payoff for all this work that we have done, so follow along with me here. I'm going to type in `rdd = sc.textFile("README.md")`, with a capital `F` in `textFile`Â â case does matter. Again, if your version of Spark has a `changes.txt` instead, just use `changes.txt` there:

Make sure you get that exactly right; remember those are parentheses, not brackets. What this is doing is creating something called a **Resilient Distributed Data store** (**rdd**), which is constructed by each line of input text in that `README.md` file. We're going to talk about rdds a lot more shortly. Spark can actually distribute the processing of this object through an entire cluster. Now let's just find out how many lines are in it and how many lines did we import into that rdd. So type in `rdd.count()` as shown in the following screenshot, and we'll get our answer. It actually ran a full-blown Spark job just for that. The answer is `104` lines in that file:

Now your answer might be different depending on what version of Spark you installed, but the important thing is that you got a number there, and you actually ran a Spark program that could do that in a distributed manner if it was running on a real cluster, so congratulations! Everything's set up properly; you have run your first Spark program already on Windows, and now we can get into how it's all working and doing some more interesting stuff with Spark. So, to get out of this Command Prompt, just type in `quit()`, and once that's done, you can close this window and move on. So, congratulations, you got everything set up; it was a lot of work but I think it's worth it. You're now set up to learn Spark using Python, so let's do it.

Installing the MovieLens movie rating dataset

The last thing we have to do before we start actually writing some code and analyzing data using Spark is to get some data to analyze. There's some really cool movie ratings data out there from a site called `grouplens.org`. They actually make their data publicly available for researchers like us, so let's go grab some. I can't actually redistribute that myself because of the licensing agreements around it, so I have to walk you through actually going to the grouplens website and downloading its `MovieLens` dataset onto your computer, so let's go get that out of the way right now.

If you just go to grouplens.org, you should come to this web page:

This is a collection of movie ratings data, which has over 40 million movie ratings available in the complete dataset, so this qualifies as big data. The way it works is that people go to MovieLens.org, shown as follows, and rate movies that they've seen. If you want, you can create an account there and play with it yourself; it's actually pretty good. I've had a go myself: Godfather, meh, not really my cup of tea. Casablanca - of course, great movie. Four and a half stars for that. Spirited Away - love that stuff:

The more you rate the better your recommendations become, and it's a good way to actually get some ideas for some new movies to watch, so go ahead and play with it. Enough people have done this; they've actually amassed, like I said, over 40 million ratings, and we can actually use that data for this book. At first, we will run this just on your desktop, as we're not going to be running Spark on a cluster until later on. For now, we will use a smaller dataset, so click on the **`datasets`** tab on grouplens.org and you'll see there are many different datasets you can get. The smallest one is 100,000 ratings:

One thing to keep in mind is that this dataset was released in 1998, so you're not going to see any new movies in here. Movies such as Star Wars, some of the Star Trek movies, and some of the more popular classics, will all be in there, so you'll still recognize some of the movies that we're working with. Go ahead and click on the `ml-100k.zip` file to download that data, and once you have that, go to your `Downloads` folder, right-click on the folder that came down, and extract it:

You should end up with a `ml-100k` folder as shown here:

Now remember, at the beginning of this chapter we set up a home for all of your stuff for this book in the `C` folder in `SparkCourse`. So navigate to your `C:\SparkCourse` directory now, and copy the `ml-100k` folder into your `C:\SparkCourse` folder:

This will give you everything you need. Now, inside this `ml-100k` folder, you should see something like the screenshot shown as follows. There's a `u.data` file that contains all the actual movie ratings and a `u.item` file that contains the lookup table for all the movie IDs to movie names. We will use both of these files extensively in the chapters coming up:

At this point, you should have an `ml-100k` folder inside your `SparkCourse` folder.

All the housekeeping is out of the way now. You've got Spark set up on your computer running on top of the JDK in a Python development environment, and we have some data to play with from `MovieLens`, so let's actually write some Spark code.

Run your first Spark program - the ratings histogram example

We just installed 100,000 movie ratings, and we now have everything we need to actually run some Spark code and get some results out of all this work that we've done so far, so let's go ahead and do that. We're going to construct a histogram of our ratings data. Of those 100,000 movie ratings that we just installed, we want to figure out how many are five star ratings, how many four stars, three stars, two stars, and one star, for example. It's really easy to do. The first thing you need to do though is to download the `ratings-counter.py` script from the download package for this book, so if you haven't done that already, take care of that right now. When we're ready to move on, let's walk through what to do with that script and we'll actually get a payoff out of all this work and then run some Spark code and see what it does.

Examining the ratings counter script

You should be able to get the `ratings-counter.py` file from the download package for this book. When you've downloaded the ratings counter script to your computer, copy it to your `SparkCourse` directory:

Once you have it there, if you've installed Enthought Canopy or your preferred PythonÂ development environment, you should be able to just double-click on that file, and up comes Canopy or your Python development environment:

```
from pyspark import SparkConf, SparkContext
import collections

conf =
SparkConf().setMaster("local").setAppName("RatingsHistogram")
sc = SparkContext(conf = conf)

lines = sc.textFile("file:///SparkCourse/ml-100k/u.data")
ratings = lines.map(lambda x: x.split()[2])
```

```
result = ratings.countByValue()

sortedResults = collections.OrderedDict(sorted(result.items()))
for key, value in sortedResults.items():
    print("%s %i" % (key, value))
```

Now I don't want to get into too much detail as to what's actually going on in the script yet. I just want you to get a little bit of a payoff for all the work you've done so far. However, if you look at the script, it's actually not that hard to figure out. We're just importing the Spark stuff that we need for Python here:

```
from pyspark import SparkConf, SparkContext
import collections
```

Next, we're doing some configurations and some set up of Spark:

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

In the next line, we're going to load the `u.data` file from the `MovieLens` dataset that we just installed, so that is the file that contains all of the 100,000 movie ratings:

```
lines = sc.textFile("file:////SparkCourse/m1-100k/u.data")
```

We then parse that data into different fields:

```
ratings = lines.map(lambda x: x.sploit()[2])
```

Then we call a little function in Spark called `countByValue` that will actually split up that data for us:

```
result = ratings.countByValue()
```

What we're trying to do is create a histogram of our ratings data. So we want to find out, of all those 100,000 ratings, how many five star ratings there are, how many four star, how many three star, how many two star and how many one star. Back at the time that they made this dataset, they didn't have half star ratings; there was only one, two, three, four, and five stars, so those are the choices. What we're going to do is count up how many of each ratings type exists in that dataset. When we're done, we're just going to sort the results and print them out in these lines of code:

```
sortedResults = collections.OrderedDict(sorted(result.items()))
for key, value in sortedResults.items():
    print("%s %i" % (key, value))
```

So that's all that's going on in this script. So let's go ahead and run that, and see if it works.

Running the ratings counter script

If you go to the **`Tools`** menu in Canopy, you have a shortcut there for Command Prompt that you can use, or you can open up Command Prompt anywhere. When you open that up, just make sure that you get into your `SparkCourse` directory where you actually downloaded the script that we're going to be using. So, type in `C:\SparkCourse` (or navigate to the directory if it's in a different location) and then type `dir` and you should see the contents of the directory. The `ratings-counter.py` and `ml-100k` folders should both be in there:

All I need to do to run it, is type in `spark-submit ratings-counter.py`-follow along with me here:

I'm going to hit ***Enter*** and that will let me run this saved script that I wrote for Spark. Off it goes, and we soon get our results. So it made short work of those 100,000 ratings. 100,000 ratings doesn't constitute really big data but we're just playing around on our desktop for now:

The results are kind of interesting. It turns out that the most common rating is four star, so people are most generous with four star ratings, with 34,000 of them in the dataset, and people seem to reserve one stars for the worst of the worst, only about 6,000 one star ratings out of our 100,00 ratings. It might be fun to go and see what actually got rated one star if you want to find some really bad movies to watch.

Summary

You ran a Spark script on your desktop, doing some real data analysis of real movie ratings data from real people; how cool is that? We just analyzed a hundred thousand movie ratings data in just a couple of seconds really and got pretty nifty results. Let's move on and start to understand what's actually happening under the hood with Spark here, how it all works and how it fits together, and then we'll go back and study this example in a little bit more depth. In the next chapter, I'll review some of the basics of Spark, go over what it's used for, and give you some of that fundamental knowledge that you need to understand how this ratings counter script actually works.

Chapter 2. Spark Basics and Spark Examples

The high-level introduction to Spark in this chapter will help you understand what Spark is all about, what's it for, who uses it, why is it so popular, and why is it so hot. Let's explore.

What is Spark?

According to Apache, Spark is *a fast and general engine for large-scale data processing*. This is actually a really good summary of what it's all about. If you have a really massive dataset that can represent anything - weblogs, genomics data, you name it - Spark can slice and dice that data up. It can distribute the processing among a huge cluster of computers, taking a data analysis problem that's just too big to run on one machine and divide and conquer it by splitting it up among multiple machines.

Spark is scalable

The way that Spark scales data analysis problems is, it runs on top of a cluster manager, so your actual Spark scripts are just everyday scripts written in Python, Java, or Scala; they behave just like any other script. Your "driver program" is what we call it, and it will run on your desktop or on one master node of your cluster. However, under the hood, when you run it, Spark knows how to take the work and actually farm it out to different computers on your cluster or even different CPUs on the same machine. Spark can actually run on top of different cluster managers. It has its own built-in cluster...

The Resilient Distributed Dataset (RDD)

In this section, we'll stop being all high level and hand-wavy and go into a little bit more depth about how Spark works from a technical standpoint. In Spark, under the hood, there's something called the Resilient Distributed Dataset object, which is like a core object that everything in Spark revolves around. Even for the libraries built on top of Spark, such as Spark SQL or MLlib, you're also using RDDs under the hood or extensions to the RDD objects to make it look like something a little bit more structured. If you understand what an RDD is in Spark, you've come ninety per cent of the way to understanding Spark.

What is the RDD?

Let's talk about the RDD in a reverse order because I'm weird like that. So, fundamentally, the RDD is a dataset, and it is an abstraction for a giant set of data, which is the main thing you need to know as a developer. What you'll do is to set up RDD objects and the RDD will load them up with big datasets and then call various methods on the RDD objects to distribute the processing of that data. Now the beauty is that although RDDs are both "distributed" and...

Ratings histogram walk-through

Remember the `RatingsHistogram` code that we ran for your first Spark program? Well, let's take a closer look at that and figure out what's actually going on under the hood with it. Understanding concepts is all well and good, but nothing beats looking at some real examples. Let's go back to the `RatingsHistogram` example that we started off with in this book. We'll break it down and understand exactly what it's doing under the hood and how it's using our RDDs to actually get the results for the `RatingsHistogram` data.

Understanding the code

The first couple of lines are just boilerplate stuff. One thing you'll see in every Python Spark script is the import statement to import `SparkConf` and `SparkContext` from the `pyspark` library that Spark includes. You will, at a minimum, need those two objects:

```
from pyspark import SparkConf, SparkContext
import collections
```

`SparkContext`, as we talked about earlier, is the fundamental starting point that the Spark framework gives you to create RDDs from. You can't create `SparkContext` without `SparkConf`, which allows you to configure the `SparkContext` and tell it things such...

Key/value RDDs and the average friends by age example

A powerful thing to do with RDDs is to put more structured data into it. One thing we can do is put key/value pairs of information into Spark RDDs and then we can treat it like a very simple database, if you will. So let's walk through an example where we have a fabricated social network set of data, and we'll analyze that data to figure out the average number of friends, broken down by age of people in this fake social network. We'll use key/value pairs and RDDs to do that. Let's cover the concepts, and then we'll come back later and actually run the code.

Key/value concepts - RDDs can hold key/value pairs

RDDs can hold key/value pairs in addition to just single values. In our previous examples, we looked at RDDs that included lines of text for an input data file or that contained movie ratings. In those cases, every element of the RDD contained a single value, either a line of text or a movie rating, but you can also store more structured information in RDDs. There's a special case where you have a list of two items contained in an RDD and that is considered a key/value pair...

Running the average friends by age example

Okay, let's make it real, let's actually get some real code and some real data and analyze the average number of friends by age in our fabricated dataset here, and see what we come up with.

At this point, you should go to the download package for this book, if you haven't already, and download two things: one is the `friends-by-age` Python script, and the other is the `fakefriends.csv` file, which is my randomly generated data that's completely fictitious, but useful for illustration. So go take care of that now. When you're done, move it into your `C:\SparkCourse` folder or wherever you're installing stuff for this course. At this point in the course, your `SparkCourse` folder should look like this:

At this moment, we need `friends-by-age.py` and `fakefriends.csv`, so let's double-click on the `friends-by-age.py` script, and Enthought Canopy or your Python environment of choice should come up. Here we have it:

Examining the script

So let's just review again what's going on here. We start off with the usual boilerplate stuff. We import what we need from `pyspark` for Spark:

```
from pyspark import SparkConf,...
```

Filtering RDDs and the minimum temperature by location example

---

Now we're going to introduce the concept of filters on RDDs, a way to strip down an RDD into the information we care about and create a smaller RDD from it. We'll do this in the context of another real example. We have some real weather data from the year 1800, and we're going to find out the minimum temperature observed at various weather stations in that year. While we're at it, we'll also use the concept of key/value RDDs as well as part of this exercise. So let's go through the concepts, walk through the code and get started.

What is filter()

Filter is just another function you can call on a mapper, which transforms it by removing information that you don't care about. In our example, the raw weather data actually includes things such as minimum temperatures observed and maximum temperatures for every day, and also the amount of precipitation observed for every day. However, all we care about for the problem we're trying to solve is the minimum temperature observed at a given station for the year. So we need to apply a filter that just looks for weather data types...

Running the minimum temperature example and modifying it for maximums

Let's see this filter in action and find out the minimum temperature observed for each weather station in the year 1800. Go to the download package for this bookÂ and download two things: the `min-temperatures` Python script and the `1800.csv` data file, which contains our weather information. Go ahead and download these now. When you're done, place them into your `C:SparkCourse` folder or wherever you're storing all the stuff for this course:

When you're ready, go ahead and double-click on `min-temperatures.py` and open that up in your editor. I think it makes a little bit more sense once you see this all together. Feel free to take some time to wrap your head around it and figure out what's going on here and then I'll walk you through it.

Examining the min-temperatures script

We start off with the usual boilerplate stuff, importing what we need from `pyspark` and setting up a `SparkContext` object that we're going to call `MinTemperatures`:

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("MinTemperatures")
sc =...
```

Running the maximum temperature by location example

I hope you did your homework. You should have had a crack at finding the maximum temperature for the year for each weather station instead of a minimum temperature, using our `min-temperatures` Python script as a starting point. If you haven't, go give it a try! Really, the only way you're going to learn this stuff is by diving in there and messing with the code yourself. I very strongly encourage you to give this a try-it's not hard. If you have done that though, let's move forward and take a look at my results. We can compare that to yours and see if you got it right.

Hopefully, you didn't have too much of a hard time figuring out the maximum temperature observed at the each weather station for the year 1800; it just involved a few changes. If you go to the download package for this book, you can download my solution to it, which is the `max-temperatures` script. If you like, you can throw that into your `SparkCourse` directory and compare your results to mine:

Let's open that up. Alright, so you can see I didn't change the code a whole lot:

```
from pyspark import SparkConf, SparkContext
...
```

Counting word occurrences using flatmap()

We'll do a really common Spark and MapReduce example of dealing with a book or text file. We'll count all the words in a text file and find out how many times each word occurs within that text. We'll put a little bit of twist on this task and work our way up to doing more and more complex twists later on. The first thing we need to do is go over the difference again between `map` and `flatMap`, because using `flatMap` and Spark is going to be the key to doing this quickly and easily. Let's talk about that and then jump into some code later on and see it in action.

Map versus flatmap

For the next few sections in this book, we'll look at your standard "count the words in a text file" sample that you see in a lot of these sorts of books, but we're going to do a little bit of a twist. We'll work our way up from a really simple implementation of counting the words, and keep adding more and more stuff to make that even better as we go along. So, to start off with, we need to review the concept of `map` versus `flatMap`. We talked about this a little bit in the past, but in our examples so far we've just...

Improving the word-count script with regular expressions

The main problem with the initial results from our `word-count` script is that we didn't account for things such as punctuation and capitalization. There are fancy ways to deal with that problem in text processing, but we're going to use a simple way for now. We'll use something called regular expressions in Python. So let's look at how that works, then run it and see it in action.

Text normalization

In the previous section, we had a first crack at counting the number of times each word occurred in our book, but the results weren't that great. We had each individual word that had different capitalization or punctuation surrounding it being counted as a word of its own, and that's not what we want. We want each word to be counted only once, no matter how it's capitalized or what punctuation might surround it. We don't want duplicate words showing up in there. There are toolkits you can get for Python such as **NLTK** (**Natural Language Toolkit**), that have fancy, very complicated ways of normalizing text data like this and automatically figuring out what words are the same. However,...

Sorting the word count results

Okay, let's do one more round of improvements on our `word-count` script. We need to sort our results of `word-count` by something useful. Instead of just having a random list of words associated with how many times they appear, what we want is to see the least used words at the beginning of our list and the most used words at the end. This should give us some actually interesting information to look at. To do this, we're going to need to manipulate our results a little bit more directly-we can't just cheat and use `countByValue` and call it done.

Step 1 - Implement countByValue() the hard way to create a new RDD

So the first thing we're going to do is actually implement what `countByValue` does by hand, the hard way. This way we can actually play with the results more directly and stick the results in an RDD instead of just getting a Python object that we need to deal with at that point. The way we do that is we take our map of words-`words.map`-and we use a mapper that just converts each individual word into that word and the value of `1-(lambda x: (x, 1))`:

```
wordCounts = words.map(lambda x: (x,...
```

Find the total amount spent by customer

At this point in the book, I think you've seen enough examples and had enough concepts that I can finally set you loose and have you try to write your very own Spark script from scratch. I realize this might be your first Python script ever, so I'm going to keep it pretty easy and I'm going to give you a lot of tips on how to be successful with it. Don't be afraid! Let me introduce the problem at hand and give you some tips you need for success, and we'll set you loose.

Introducing the problem

I'm going to start you off with a pretty simple example here just to get your feet wet. What you're going to do is go to the download package for this book and find the `customerorders.csv` file. This just contains some random fake data that I generated. The input data in that file is going to look like this:

We have comma-separated fields of a customer ID, an item ID, and the amount spent on that item. What I want you to do is write a Spark script that consolidates that down to the total amount spent by customer ID. In the following example, we see the customer ID 44 appears twice and bought two items,...

Check your results and sort them by the total amount spent

Did you do your homework? I hope so. Hopefully, you were able to draw on our previous examples and now have a running script that adds up the total amount spent by customer in my little fake dataset. Let's compare your implementation to my implementation and your results to mine and see if they match up. If they don't, hopefully you'll learn something from it.

Let's have a look at my solution. If you go to the download package for this book, you will see my `total-spent-by-customer` script, feel free to download that and put it into your `SparkCourse` folder alongside your own solution to this problem and the `book.txt` file that you should have from the previous section. Go ahead and open that up. When you're ready, here's my take at it:

This is a very simple script, but the idea was just to get you into the practice of writing your own Spark programs. Even if it's a simple one, writing your first program is a big accomplishment. Let's go through how I did it and if you had any trouble with it yourself, you can compare your code with mine and see what might be different.

We have...

Check your sorted implementation and results against mine

Let's take a look at my implementation of sorting the results for the total amount spent by customer and compare it with yours. If you got to this point and you haven't actually tried this yourself or you got stuck, let me give you one more hint before I show you my solution. If you look at the `word-count-better-sorted` script that we used in the previous example for doing word frequency counts sorted by word frequency, you'll see that we did something very similar in there. If you got stuck, have a look at that script and give it another try first before you peek at the answer here.

If you've given it a try, go ahead and download the `total-spent-by-customer-sorted` script from the download package for this bookÂ and compare your solution to mine. There's more than one way to do these things, so just because I did it one way it doesn't mean you can't do it another way. As long as it works, that's what matters:

You can see all I really did was add a couple of different lines. Take a look at my `totalByCustomer` RDD here, which is just the key/value RDD of customer IDs to the...

Summary

We've covered a lot of ground in this chapter, and I hope it's given you an idea of the kinds of things that you can do with Spark and the power that it gives you. Please do continue to explore and experiment with these examples, altering things to see how they function and gaining familiarity with the workings of Spark. In the next chapter, we're going to turn our attention to the cloud and start working with really big data when we find out how to run Spark on a cluster.

Chapter 3. Advanced Examples of Spark Programs

We'll now start working our way up to some more advanced and complicated examples with Spark. Like we did with the `word-count` example, we'll start off with something pretty simple and just build upon it. Let's take a look at our next example, in which we'll find the most popular movie in our MovieLens dataset.

Finding the most popular movie

Let's start by reviewing the data format of the `MovieLens` dataset, the `u.data` file.

As you might recall, the `u.data` file on each line, consists of a user ID, a movie ID, a rating, and a timestamp. Each line says, "this user watched this movie, gave it this rating, and did it at this time":

Our task is to just figure out which movie was watched most often or which movie ID appears most frequently in the entire dataset. This isn't a very hard thing to do; in fact, if you want to go give it a crack yourself, feel free. In this section we'll take a look at the implementation that I came up with, get that to run, and see what we come up with

Examining the popular-movies script

In the download package for this book, you'll find a `popular-movies` Python script. Download that, put it in your `SparkCourse` folder, and open it up. We will print out a list of all of the movies and the number of times they appear, and then sort them based on how often they appear. It's a pretty simple script here, there's nothing really new in this example, but there will be when we build upon it:

```
from pyspark import SparkConf,...
```

Using broadcast variables to display movie names instead of ID numbers

In this section, we'll figure out how to actually include information about the names of the movies in our `MovieLens` dataset with our Spark job. We'll include them in such a way that they'll get broadcast out to the entire cluster. I'm going to introduce a concept called broadcast variables to do that. There are a few ways we could go about identifying movie names; the most straightforward method would be to just read in the `u.item` file to look up what movie ID 50 was, see it means *Star Wars*, load up a giant table in Python, and reference that when we're printing out the results within our driver program at the end. That'd be fine, but what if our executors actually need access to that information? How do we get that information to the executors? What if one of our mappers, or one of our reduce functions or something, needed access to the movie names? Well, it turns out that Spark will sort of automatically and magically forward any Python objects that you might need for your mappers or reducers to complete. This can be very handy, but it can also be very...

Finding the most popular superhero in a social graph

Believe it or not, there's actually a publicly available dataset, where someone figured out all the appearances that Marvel superheroes had with each other in different comic books. We can use that to figure out relationships between superheroes in the Marvel Universe and other aspects of them that we might find interesting. Let's start off with a simple example, where we just try to find the most popular superhero in the Marvel dataset. Is Spider-Man or The Hulk the most commonly appearing superhero in Marvel Comics? Well, the answer might surprise you. Let's go dig in and find out.

Superhero social networks

We have a superhero social network that someone constructed, just by looking at what comic books characters appeared in together. So the idea is that if a comic book character appeared together with another comic book character in the same comic book, they're considered to be friends, they have a co-occurrence and they are therefore connected. You may have characters connected in complex patterns. This is really like a social graph that we're dealing with. Let's say the Hulk...

Running the script - discover who the most popular superhero is

Let's dive into the code for finding the most popular superhero in the Marvel Universe and get our answer. Who will it be? We'll find out soon. Go to the download package for this bookÂ and you're going to download three things: the `Marvel-graph.txt` data file, which contains our social network of superheroes, the `Marvel-names.txt` file, which maps superhero IDs to human-readable names, and finally, the `most-popular-superhero` script. Download all that into your `SparkCourse` folder and then open up `most-popular-superhero.py` in your Python environment:

Alright, let's see what's going on here. We have the usual stuff at the top so let's get down to the meat of it

Mapping input data to (hero ID, number of co-occurrences) per line

The first thing we do, if you look at line 14, is load up our `names.txt` file into an RDD called `names` using `sc.textFile`:

```
names = sc.textFile("file:///SparkCourse/marvel-names.txt")
```

We're going to do this name lookup a little bit differently than we did in the previous example. Instead of using a broadcast variable, we'll store the superhero ID to...

Superhero degrees of separation - introducing the breadth-first search algorithm

You might have heard how everyone is connected through six degrees of separation. Somebody you know knows somebody else who knows somebody else and so on; eventually, you can be connected to pretty much everyone on the planet. Or maybe you've heard about how Kevin Bacon is within a few degrees of separation of pretty much everybody in Hollywood. Well, I used to work at imdb.com, and I can tell you that is true, Kevin Bacon is pretty well connected, but a lot of other actors are too. Kevin Bacon is actually not the most connected actor, but I digress! We want to bring this concept of degrees of separation to our superhero dataset, where we have this virtual social network of superheroes.

Let's figure out the degrees of separation between any two superheroes in that dataset. Is the Hulk connected to Spider-Man closely? How do you find how many connections there are between any two given superheroes that we have? To find the answer, we need to introduce an algorithm called breadth-first search. It's a "computer sciencey," "network theory" sort of thing,...

Accumulators and implementing BFS in Spark

Now that we have the concept of breadth-first-search under our belt and we understand how that can be used to find the degrees of separation between superheroes, let's apply that and actually write some Spark code to make it happen. So how do we turn breadth-first search into a Spark problem? This will make a lot more sense if that explanation of how BFS works is still fresh in your head. If it's not, it might be a good idea to go back and re-read the previous section; it will really help a lot if you understand the theory.

Convert the input file into structured data

The first thing we need to do is actually convert our data file or input file into something that looks like the nodes that we described in the BFS algorithm in the previous section, *Superhero degrees of separation - introducing breadth-first search*.

We're starting off, for example, with a line of input that looks like the one shown here that says hero ID `5983` appeared with heroes `1165`, `3836`, `4361`, and `1282`:

We want to convert that to something a little bit more interesting and useful. We want to convert that to structured data,...

Superhero degrees of separation - review the code and run it

Using breadth-first search, let's actually find the degrees of separation between two given superheroes in our Marvel superhero dataset. In the download package for this book, download the `degrees-of-separation` script into your `SparkCourse` folder. We'll work up a pretty good library here of different examples, so keep this handy. There's a good chance that some problem you face in the future will have a similar pattern to something we've already done here, and this might be a useful reference for you. Once you have downloaded that script, double-click it. We already have the `Marvel-graph` and `Marvel-names` text files for our input from previous sections.

Here is the `degrees-of-separation` script:

The point here is just to illustrate how problems that may not seem like they lend themselves to Spark at first, actually can be incremented in Spark with a little bit of creative thinking. I also want to introduce the concept of accumulators here as we go. So let's look at our code. We have the usual boilerplate stuff at the top, nothing changed there:

```
#Boilerplate stuff:
from...
```

Item-based collaborative filtering in Spark, cache(), and persist()

We're now going to cover a topic that's near and dear to my heart-collaborative filtering. Have you ever been to some place like amazon.com and seen something like "people who bought this also bought," or have you seen "similar movies" suggested on imdb.com? I used to work on that. In this section, I'm going to show you some general algorithms on how that works under the hood. Now I can't tell you exactly how Amazon does it, because Jeff Bezos would hunt me down and probably do terrible things to me, but I can tell you some generally known techniques that you can build upon for doing something similar. Let's talk about a technique called item-based collaborative filtering and discuss how that works. We'll apply it to our MovieLens data to actually figure out similar movies to each other based on user ratings.

We're doing some pretty complicated and advanced stuff at this point in the book. The good news is this is probably the culmination of the book in terms of the difficulty level. Stay with me and bear with me, because after this section, things get easier,...

Running the similar-movies script using Spark's cluster manager

Leading up to this point has been a lot of work, but we now have a Spark program that should give us similar movies to each other. We can figure out what movies are similar to each other, just based on similarities between user ratings. Let's turn this movie similarities problem into some real code, run it, and look at the results. Go to the download package for this book, you will find a `movie-similarities` script. Download that to your `SparkCourse` folder and open it up. We're going to keep on using the MovieLens 100,000 rating dataset for this example, so there's no new data to download, just the script. This is the most complicated thing we're going to do in this course, so let's just get through the script and walk through what it's doing. We described it at a high level in the previous section, but let's go through it again.

Examining the script

You can see we're importing the usual stuff at the top of the script. We do need to import the square root function from the `math` package in Python as well, so that we can do our similarity metric and compute similarities...

Improving the quality of the similar movies example

Now it's time for your homework assignment. Your mission, should you choose to accept it, is to dive into this code and try to make the quality of our similarities better. It's really a subjective task; the objective here is to get you to roll up your sleeves, dive in, and start messing with this code to make sure that you understand it. You can modify it and get some tangible results out of your changes. Let me give you some pointers and some tips on what you might want to try here and we'll set you loose.

We used a very naive algorithm to find similar movies in the previous section with a cosine similarity metric. The results, as we saw, weren't that bad, but maybe they could be better. There are ways to actually measure the quality of a recommendation or similarity, but without getting it into that, just dive in there, try some different ideas and see what effect it has, and maybe they qualitatively will look better to you. At the end of the day, maybe that's all the matters. Here are some ideas and some simple things to try:

- Discard bad ratings:
    - Maybe you could just try...

Summary

We've covered a lot of ground in this chapter, and I hope it's given you an idea of the kinds of things that you can do with Spark and the power that it gives you. Please do continue to explore and experiment with these examples, altering things to see how they function and gaining familiarity with the workings of Spark. In the next chapter, we're going to turn our attention to the cloud and start working with really big data when we find out how to run Spark on a cluster.

Chapter 4. Running Spark on a Cluster

Now it's time to graduate off of your desktop computer and actually start running some Spark jobs in the cloud on an actual Spark cluster.

Introducing Elastic MapReduce

The easiest way to actually get up and running on a cluster, if you don't already have a Spark cluster, is using Amazon's Elastic MapReduce service. Even though it says MapReduce in the name, you can actually configure it to set up a Spark cluster for you and run that on top of Hadoop â it sets everything up for you automatically. Let's walk through what Elastic MapReduce is, how it interacts with Spark, and how to decide if it's really something you want to be messing with.

Why use Elastic MapReduce?

Using Amazon's Elastic MapReduce service is an easy way to rent the time you need on a cluster to actually run your Spark job. You don't have to just run MapReduce jobs, you can actually run Spark and use the underlying Hadoop environment to run as your cluster manager. It has something called Hadoop YARN. If you've taken my course on MapReduce and Hadoop you will have heard of this already. Basically, YARN is Hadoop's cluster manager and Spark is able to run on top of it-that is all installed as part of Elastic MapReduce for you.

Let's back up a little bit. This is a service offered by Amazon.com....

Setting up our Amazon Web Services / Elastic MapReduce account and PuTTY

To get started with Amazon Web Services, first we're going to walk through how to create an account on AWS if you haven't one already. When we're done, we're going to figure out how to actually connect to the instances that we might be spinning up on Web Services. When we create a cluster for Spark, we need a way to log in to the master node on that cluster and actually run our script there. To do so, we need to get our credentials for logging in to any instances that our Spark cluster spins up. We'll also set up a Terminal, if you're on Windows, called PuTTY, and go through how to actually use that to connect to your instances.

Okay, let's go through how to set up an Amazon Web Services account and get started with Elastic MapReduce. We'll also figure out how to connect to our instances on Elastic MapReduce. Head over to aws.amazon.com:

As I mentioned in the previous section, if you don't want to risk spending money, it might be best not to follow along on your computers at this point. You're going to need a credit card number to sign up for an account here,...

Partitioning

Now that we are running on a cluster, we need to modify our driver script a little bit. We'll look at the movie similarity sample again and figure out how we can scale that up to actually use a million movie ratings. To do so, you can't just run it as is and hope for the best, you wouldn't succeed if you were to do that. Instead, we have to think about things such as how is this data going to be partitioned? It's not hard, but it is something you need to address in your script. In this section we'll cover partitioning and how to use it in your Spark script.

Let's get on with actually running our `movie-similarities` script on a cluster. This time we're going to talk about throwing a million ratings at it instead of a hundred thousand ratings. Now, if we were to just modify our script to use the 1 million rating dataset from `grouplens.org`, it's not going to run on your desktop obviously. The main reason is that when we use `self-join` to generate every possible combination of movie pairs, it blows up like you wouldn't believe. You'll run out of memory and your job will fail pretty quickly if you just run it on your...

Creating similar movies from one million ratings - part 1

Let's modify our `movie-similarities` script to actually work on the 1 million ratings dataset and make it so it can run in the cloud on Amazon Elastic MapReduce, or any Spark cluster for that matter. So, if you haven't already, go grab the `movie-similarities-1m` Python script from the download package for this book, and save it wherever you want to. It's actually not that important where you save this one because we're not going to run it on your desktop anyway, you just need to look at it and know where it is. Open it up, just so we can take a peek, and I'll walk you through the stuff that we actually changed:

Changes to the script

Now, first of all, we made some changes so that it uses the 1 million ratings dataset from Grouplens instead of the 100,000 ratings dataset. If you want to grab that, go over to grouplens.org and click on **`datasets`**:

You'll find it in the **`MovieLens 1M Dataset`**:

This data is a little bit more current, it's from 2003. They do have a current dataset that is updated as of this month, but you're going to need a pretty large cluster to handle the 40 million...

Creating similar movies from one million ratings - part 2

Now it's time to run our similarities script on a Spark cluster in the cloud on Elastic MapReduce. This is a pretty big deal, it's kind of the culmination of the whole course here, so let's kick it off and see what happens.

Our strategy

Before we actually run our script on a Spark cluster using Amazon's Elastic MapReduce service, let's talk about some of the basic strategies that we're going to use to do that.

Specifying memory per executor

Like we talked about earlier, we're going to use the default empty `SparkConf` in the driver script. That way we'll use the defaults that Elastic MapReduce sets up for us, and that will automatically tell Spark that it should be running on top of EMR's Hadoop cluster manager. Then it will automatically know what the layout of the cluster is, who the master is, how many client machines I have, who they are, how many executors they have, and so on. Now, when we're actually running this, we're going to pass one extra argument to `spark-submit`. In the past, we always just called `spark-submit` followed by the script name and whatever parameters we...

Creating similar movies from one million ratings â part 3

About 15 minutes after I set off our `movie-similarities-1m` script on a cluster using EMR, I have some actual results to look at. Let's review what happened.

Assessing the results

Here are the results:

The top similar movie to *Star Wars* Episode Four, was *Star Wars* Episode Five, not too surprising. But the next entry is a little bit surprising, some little movie called *Sanjuro* had a very high similarity score. Let's look at what's going on there. Its actual strength, the number of people that rated that together with *Star Wars*, was only 60, so I think it's safe to say that is kind of a spurious result. Now that we're using a million ratings, we probably need to increase that minimum threshold on the number of co-raters in order to actually display a result. By doing so, we could probably pretty easily filter out movies like that and instead get *Raiders of the Lost Ark* as our second result instead of as our third. I think the position of *Raiders of the Lost Ark* is interesting because it's actually scoring better than *Return Of The Jedi*. I'm no film critic, but maybe you too...

Troubleshooting Spark on a cluster

So let's start talking about what we do when things go wrong with our Spark job. It has a web-based console that we can look at in some circumstances, so let's start by talking about that.

Troubleshooting Spark jobs on a cluster is a bit of a dark art. If it's not immediately obvious what is going on from the output of the Spark driver script, a lot of times what you end up doing is throwing more machines at it and throwing more memory at it, like we looked at with the executor memory option. But if you're running on your own cluster or one that you have within your own network, Spark does offer a console UI that runs by default on port `4040`. It does give you a little bit more of a graphical, in-depth look as to what's going on and a way to access the logs and see which executor is doing what. This can be helpful in understanding what's happening. Unfortunately, in Elastic MapReduce, it's pretty much next to impossible to connect to Spark's UI console from outside Amazon's network.

However, but if you have your own cluster running on your own network, it might be a good option for you, so you can...

More troubleshooting and managing dependencies

In this section, I want to talk about a few more troubleshooting tips with Spark. There are weird things that will happen and have happened to me in the past when working with Spark. It's not always obvious what to do about them, so let me impart some of my experience to you here. Then we'll talk about managing code dependencies within Spark jobs as well.

Troubleshooting

So let's talk about troubleshooting a little bit more. I can tell you, I did need to do some troubleshooting to get that million ratings job running successfully on my Spark cluster. We'll start by talking about logs. Where are the logs? We saw some stuff scroll by from the driver script, and in practice, if you're running on EMR, that's pretty much all you'll have to go on. Now, as I showed you, if you're in standalone mode and you have access directly, on the network to your master node, all the log information is displayed in this beautiful graphical form in the web UI. However, when you're running on top of YARN, those logs end up getting distributed and you need to collect them after the fact using this command:

Summary

This concludes the part of the course about Spark core, covering the things you can do with Spark itself. We did pretty much everything there is to do, and we actually ran a million ratings, and analyzed on a real cluster in the cloud using Spark. So congratulations, if you've got this far! You are now pretty knowledgeable about Spark. In the next chapter, we'll talk about some of the technologies built on top of Spark that are still part of this greater Spark package.

Chapter 5. SparkSQL, DataFrames, and DataSets

In this chapter, we'll spend some time talking about SparkSQL. This is becoming an increasingly important part of Spark; it basically lets you deal with structured data formats. This means that instead of the RDDs that contain arbitrary information in every row, we're going to give the rows some structure. This will let us do a lot of different things, such as treat our RDDs as little databases. So, we're going to call them DataFrames and DataSets from now on, and you can actually perform SQL queries and SQL-like operations on them, which can be pretty powerful.

Introducing SparkSQL

What is structured data? Basically, it means that when we extend the concept of an RDD to a DataFrame object, we provide the data in the RDD with some structure.

One way to think of it is that it's fundamentally an RDD of row objects. By doing this, we can construct SQL queries. We can have distinct columns in these rows, and we can actually form SQL queries and issue commands in a SQL-like style, which we'll see shortly. Because we have an actual schema associated with the DataFrame, it means that Spark can actually do even more optimization than what it normally would. So, it can do query optimization, just like you would on a SQL database, when it tries to figure out the optimal plan for executing your Spark script. Another nice thing is that you can directly read and write to JSON files or JDBC-compiled and compliant databases. This means that if you do have your source data that's already in a structured format, for example, inside a relational database or inside JSON files, you can import it directly into a DataFrame and treat it as if it were a SQL database. So, powerful stuff, isn't it?

Using SparkSQL...

Executing SQL commands and SQL-style functions on a DataFrame

Alright, open up the `sparksql.py` file that's included in the download files for this book. Let's take a look at it as a real-world example of using SparkSQL in Spark 2.0. You should see the following code in your editor:

Notice that we're importing a few things here. We're importing the `SparkSession` object and the `Row` object. The `SparkSession` object is basically Spark 2.0's way of creating a context to work with SparkSQL. We'll also import collections here:

```
from pyspark.sql import SparkSession
from pyspark.sql import Row

import collections
```

Earlier, we used to create `sparkContext` objects, but now, we'll create a `SparkSession` object:

```
# Create a SparkSession (Note, the config section is only for
Windows!)
spark = SparkSession.builder.config("spark.sql.warehouse.dir",
"file:///C:/temp").appName("SparkSQL").getOrCreate()
```

So what we're doing here is creating something called `spark` that's going to be a `SparkSession` object. We'll use a `builder` method on it, then we will just combine these different parameters to actually get a session that we can work with. This will give us...

Using DataFrames instead of RDDs

Just to drive home how you can actually use DataFrames instead of RDDs, let's go through an example of actually going to one of our earlier exercises that we did with RDDs and do it with DataFrames instead. This will illustrate how using DataFrames can make things simpler. We go back to the example where we figured out the most popular movies based on the MovieLens DataSet ratings information. If you want to open the file, you'll find it in the download package as `popular-movies-dataframe.py`, or you can just follow along typing it in as you go. This is what your file should look like if you open it in your IDE:

Let's go through this in detail. First comes our import statements:

```
from pyspark.sql import SparkSession
from pyspark.sql import Row
from pyspark.sql import functions
```

We start by importing `SparkSession`, which again is our new API in Spark 2.0 for doing DataFrame and DataSet operations. We will import the `Row` class and `functions`, so we can do SQL functions on them:

Next, we have our `loadMovieNames` function:

```
def loadMovieNames():
    movieNames = {}
    with open("ml-100k/u.ITEM") as f:
    ...
```

Summary

It is interesting how you can actually use these high-level APIs using SparkSQL to save on coding. For example, just look at this one line of code:

```
topMovieIDs =
movieDataset.groupBy("movieID").count().orderBy("count",
ascending=False).cache()
```

Remember that to do the same thing earlier, we had to kind of jump through some hoops and create key/value RDDs, reduce the RDD, and do all sorts of things that weren't very intuitive. Using SparkSQL and DataSets, however, you can do these exercises in a much more intuitive manner. At the same time, you allow Spark the opportunity to represent its data more compactly and optimize those queries in a more efficient manner.

Again, DataFrames are the way of the future with Spark. If you do have the choice between using an RDD and a DataFrame to do the same problem, opt for a DataFrame. It is not only more efficient, but it will also give you more interoperability with more components within Spark going forward. So there you have it: Spark SQL DataFrames and DataSets in a nutshell and in action. Remember, this lesson is very important going forward. Let's move on and take a look at some of...

Chapter 6. Other Spark Technologies and Libraries

Introducing MLlib

If you're doing any real data or science data mining or machine learning stuff with Spark, you're going to find the MLlib library very helpful. **MLlib** (**machine learning library**) is built on top of Spark as part of the Spark package. It contains some useful libraries for machine learning and data mining and some functions that you might find helpful. Let's review what some of those are and take a look at them. When we're done, we'll actually use MLlib to generate movie recommendations for users using the MovieLens dataset again.

MLlib capabilities

The following is a list of different features of MLlib. They have support in the library to help you with these various techniques:

- Feature extraction
  - Term Frequency / Inverse Document frequency useful for search
- Basic statistics
  - Chi-squared test, Pearson or Spearman correlation, min, max, mean, and variance
- Linear regression and logistic regression
- Support Vector Machines
- NaÃ¯ve Bayes classifier
- Decision trees
- K-Means clustering
- Principal component analysis and singular value decomposition
- Recommendation using Alternating Least Squares

I don't really have time to go into what...

Using MLlib to produce movie recommendations

Let's take a look at some code to actually run Alternating Least Squares recommendations on the MovieLens dataset. You'll see just how simple it is to do and we'll take a look at the results.

You can download the script from the download package for this book. Look for `movie-recommendations-als.py`, download that into your `SparkCourse` folder, and then we can play with it. This is going to require us to input a user ID that I want recommendations for. So, how do we know if recommendations are good? Since we don't personally know any of the people that are in this dataset from MovieLens, we need to create a fictitious user; we can kind of hack their data to stick it in there. So, in the `ml-100k` folder, I've edited the `u.data` file. What I've done here is I've added three lines to the top for user ID 0, because I happen to know that user ID 0 does not exist in this dataset:

I looked up a few movies that I'm familiar with, so I can get a little more of a gut feel as to how good these recommendations might be. So, movie ID 50 is actually *Star Wars* and I've given that a five star rating. ID 172...

Analyzing the ALS recommendations results

Open up **`Command Prompt`** and type `spark-submit movie-recommendations-als.py` with user `0`:

That user `0` is my Star Wars fan that doesn't like Gone With The Wind.

Off it goes, using all the cores that I have. It should finish quite quickly. For such a fancy algorithm, that came back creepily fast, almost suspiciously so:

So, for my fictitious user who loves *Star Wars*Â and *The Empire Strikes Back*, but hated *Gone With The Wind*, the number one recommendations it produced was something called *Love in the Afternoon*Â and *Roommates*. What? What is this stuff? That's crazy. *Lost in Space*, okay, I can go with that, but the rest of this just doesn't make sense. What's worse is if I run it again, I'll actually get different results! Now, it could be that the algorithm is taking some shortcuts and randomly sampling things to save time, but even so, that's not good news.

Let's see what we get if we run it again. We get a totally different set of results:

There's something in there that I might agree with- *Army of Darkness*, yeah, people who like *Star Wars*Â might be into that. But this other stuff? Not so...

Using DataFrames with MLlib

So, back when we mentioned Spark SQL, remember I said DataFrames are kind of the way of the future with Spark and it's going to be tying together different components of Spark? Well, that applies to MLlib as well. There's a new DataFrame-based API in Spark 2.0 for MLlib, which is the preferred API going forward. The one that we just mentioned is still there if you want to keep using RDDs, but if you want to use DataFrames instead, you can do that too, and that opens up some interesting possibilities. Using DataFrames means you can import structured data from a database or JSON file or even a streaming source, and actually execute machine learning algorithms on that as it comes in. It's a way to actually do machine learning on a cluster using structured data from a database.

We'll look at an example of doing that with linear regression, and just to refresh you, if you're not familiar with linear regression, all that is fitting a line to a bunch of data. So imagine, for example, that we have a bunch of data of people's heights and weights. We imagine there's some linear relationship between these two...

Spark Streaming and GraphX

The last two things I want to cover in this chapter are Spark Streaming and GraphX, which are two technologies built on top of Spark. Spark Streaming handles continually incoming real-time data, say from a series of web logs from web servers that are running all the time. GraphX offers some tools for network analysis, kind of like the social network analysis that we were doing back when we were looking at superheroes and their relationships to each other. I'm going to cover Spark Streaming and GraphX at kind of a hand-wavey high level, because they're currently incomplete. Neither one really has good support for Python yet, right now they only work in Scala and they're in the process of being ported to Java and Python. However, by the time you read this book they might very well be available. So, all we can do is talk about them at this point, but I think you'll get the idea. So, follow along and let's see what's there.

What is Spark Streaming?

Spark Streaming is useful for very specific tasks. If you have a continual stream of data that needs constant analysis, then Spark Streaming is for you. A common...

Summary

That's Spark Streaming and GraphX, and with that, we've covered pretty much everything there is about Spark, at least as of this date. Congratulations! You are now very knowledgeable about Spark. If you've actually followed along and gone through the examples and done the exercises, I think you can call yourself a Spark developer, so well done Let's talk about where to go from here next and what the next steps are for continued learning.

Chapter 7. Where to Go From Here? – Learning More About Spark and Data Science

If you made it this far, congratulations! Thanks for sticking through this whole book with me. If you feel like it was a useful experience and you've learned a few things, please write a review on this book. That will help me improve the book in future editions. Let me know what I'm doing right and what I'm doing wrong, and it will help other prospective students understand if they should give this course a try. I'd really appreciate your feedback.

So where do you go from here? What's next? Well, there's obviously a lot more to learn. We've covered a lot of the basics of Spark, but of course there is more to the field of data science and big data as a whole. If you want more information on this topic, *Packt* offer an extensive collection of Spark books and courses. There are also some other books that I can recommend about Spark.Â I don't make any money from these, so don't worry, I'm not trying to sell you anything. I have personally enjoyed *Learning Spark,Â O'Reilly*, that IÂ found to be a useful reference while learning. It has a lot of good...