Fast Data Processing with Spark
Second Edition

Spark is a framework used for writing fast, distributed programs. Spark solves similar problems as Hadoop MapReduce does, but with a fast in-memory approach and a clean functional style API. With its ability to integrate with Hadoop and built-in tools for interactive query analysis (Spark SQL), large-scale graph processing and analysis (GraphX), and real-time analysis (Spark Streaming), it can be interactively used to quickly process and query big datasets.

Fast Data Processing with Spark - Second Edition covers how to write distributed programs with Spark. The book will guide you through every step required to write effective distributed programs, from setting up your cluster and interactively exploring the API to developing analytics applications and tuning them for your purposes.

Who this book is written for
Fast Data Processing with Spark - Second Edition is for software developers who want to learn how to write distributed programs with Spark. It will help developers who have had problems that were too big to be dealt with on a single computer. No previous experience with distributed programming is necessary. This book assumes knowledge of either Java, Scala, or Python.

What you will learn from this book
- Install and set up Spark on your cluster
- Prototype distributed applications with Spark’s interactive shell
- Learn different ways to interact with Spark’s distributed representation of data (RDDs)
- Query Spark with a SQL-like query syntax
- Effectively test your distributed software
- Recognize how Spark works with big data
- Implement machine learning systems with highly scalable algorithms

In this package, you will find:

- The author’s biography
- A preview chapter from the book, Chapter 5 'Loading and Saving Data in Spark'
- A synopsis of the book’s content
- More information on Fast Data Processing with Spark Second Edition

About the Authors

Krishna Sankar is a chief data scientist at http://www.blackarrow.tv/, where he focuses on optimizing user experiences via inference, intelligence, and interfaces. His earlier roles include principal architect, data scientist at Tata America Intl, director of a data science and bioinformatics start-up, and a distinguished engineer at Cisco. He has spoken at various conferences, such as Strata-Sparkcamp, OSCON, Pycon, and Pydata about predicting NFL (http://goo.gl/movfds), Spark (http://goo.gl/E4kqMD), data science (http://goo.gl/9pyJMH), machine learning (http://goo.gl/SXF53n), and social media analysis (http://goo.gl/D9YPVQ). He was a guest lecturer at Naval Postgraduate School, Monterey. His blogs can be found at https://doubleclix.wordpress.com/. His other passion is Lego Robotics. You can find him at the St. Louis FLL World Competition as the robots design judge.

Holden Karau is a software development engineer and is active in the open source sphere. She has worked on a variety of search, classification, and distributed systems problems at Databricks, Google, Foursquare, and Amazon. She graduated from the University of Waterloo with a bachelor's of mathematics degree in computer science. Other than software, she enjoys playing with fire and hula hoops, and welding.
Fast Data Processing with Spark  
Second Edition

Apache Spark has captured the imagination of the analytics and big data developers, and rightfully so. In a nutshell, Spark enables distributed computing on a large scale in the lab or in production. Till now, the pipeline collect-store-transform was distinct from the Data Science pipeline reason-model, which was again distinct from the deployment of the analytics and machine learning models. Now, with Spark and technologies, such as Kafka, we can seamlessly span the data management and data science pipelines. We can build data science models on larger datasets, requiring not just sample data. However, whatever models we build can be deployed into production (with added work from engineering on the "ilities", of course). It is our hope that this book would enable an engineer to get familiar with the fundamentals of the Spark platform as well as provide hands-on experience on some of the advanced capabilities.

What This Book Covers

*Chapter 1, Installing Spark and Setting up your Cluster,* discusses some common methods for setting up Spark.

*Chapter 2, Using the Spark Shell,* introduces the command line for Spark. The Shell is good for trying out quick program snippets or just figuring out the syntax of a call interactively.

*Chapter 3, Building and Running a Spark Application,* covers Maven and sbt for compiling Spark applications.

*Chapter 4, Creating a SparkContext,* describes the programming aspects of the connection to a Spark server, for example, the SparkContext.

*Chapter 5, Loading and Saving Data in Spark,* deals with how we can get data in and out of a Spark environment.

*Chapter 6, Manipulating your RDD,* describes how to program the Resilient Distributed Datasets, which is the fundamental data abstraction in Spark that makes all the magic possible.

*Chapter 7, Spark SQL,* deals with the SQL interface in Spark. Spark SQL probably is the most widely used feature.

*Chapter 8, Spark with Big Data,* describes the interfaces with Parquet and HBase.
Chapter 9, *Machine Learning Using Spark MLlib*, talks about regression, classification, clustering, and recommendation. This is probably the largest chapter in this book. If you are stranded on a remote island and could take only one chapter with you, this should be the one!

Chapter 10, *Testing*, talks about the importance of testing distributed applications.

Chapter 11, *Tips and Tricks*, distills some of the things we have seen. Our hope is that as you get more and more adept in Spark programming, you will add this to the list and send us your gems for us to include in the next version of this book!
By this point in the book, you have already experimented with the Spark shell, figured out how to create a connection with the Spark cluster, and built jobs for deployment. Now to make those jobs useful, you will learn how to load and save data in Spark. Spark's primary unit for representation of data is an RDD, which allows for easy parallel operations on the data. Other forms of data, such as counters, have their own representation. Spark can load and save RDDs from a variety of sources.

**RDDs**

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

Before jumping into the details on the supported data sources/links, take some time to learn about what RDDs are and what they are not. It is crucial to understand that even though an RDD is defined, it does not actually contain data but just creates the pipeline for it. (As an RDD follows the principle of lazy evaluation, it evaluates an expression only when it is needed, that is, when an action is called for.) This means that when you go to access the data in an RDD, it could fail. The computation to create the data in an RDD is only done when the data is referenced by caching or writing out the RDD. This also means that you can chain a large number of operations and not have to worry about excessive blocking in a computational thread. It's important to note during application development that you can write code, compile it, and even run your job; unless you materialize the RDD, your code may not have even tried to load the original data.
Loading and Saving Data in Spark

Each time you materialize an RDD, it is recomputed; if we are going to be using something frequently, a performance improvement can be achieved by caching the RDD.

Loading data into an RDD

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

One of the easiest ways of creating an RDD is taking an existing Scala collection and converting it into an RDD. The SparkContext object provides a function called parallelize that takes a Scala collection and turns it into an RDD over the same type as the input collection, as shown here:

- Scala:
  ```scala
  val dataRDD = sc.parallelize(List(1,2,4))
  dataRDD.take(3)
  ```

- Java:
  ```java
  import java.util.Arrays;
  import org.apache.spark.SparkConf;
  import org.apache.spark.api.java.
  import org.apache.spark.api.java.function.Function;

  public class LDSV01 {

    public static void main(String[] args) {
      // TODO Auto-generated method stub
      SparkConf conf = new SparkConf().setAppName("Chapter 05").setMaster("local");
      JavaSparkContext ctx = new JavaSparkContext(conf);
      JavaRDD<Integer> dataRDD = ctx.parallelize(Arrays.asList(1,2,4));
      System.out.println(dataRDD.count());
      System.out.println(dataRDD.take(3));
    }
  }
  ```

The reason for a full program in Java is that you can use the Scala and Python shell, but for Java you need to compile and run the program. I use Eclipse and add the JAR file /usr/local/spark-1.1.1/assembly/target/scala-2.10/spark-assembly-1.1.1-hadoop2.4.0.jar in the Java build path.

- Python:
  
rdd = sc.parallelize([1,2,3])
  rdd.take(3)

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

- Scala:
  
  import org.apache.spark.SparkFiles;
  
  ...  
  sc.addFile("spam.data")
  val inFile = sc.textFile(SparkFiles.get("spam.data"))
  inFile.first()

- Java:
  
  import org.apache.spark.SparkConf;
  import org.apache.spark.api.java.*;
  import org.apache.spark.SparkFiles;;

  public class LDSV02 {

    public static void main(String[] args) {
      SparkConf conf = new SparkConf().setAppName("Chapter 05").
      setMaster("local");
      JavaSparkContext ctx = new JavaSparkContext(conf);
      System.out.println("Running Spark Version : \" +ctx.version());
      ctx.addFile("/Users/ksankar/fpdb-v1ii/data/spam.data");
JavaRDD<

```java
    JavaRDD<String> lines = ctx.textFile(SparkFiles.get("spam.data"));
    System.out.println(lines.first());
```
Frequently, your input files will be CSV or TSV files, which you will want to read and parse and then create RDDs for processing. The two ways of reading CSV files are either reading and parsing them using our own functions or using a CSV library like opencsv.

Let's first look at parsing using our own functions:

- **Scala:**
  ```scala
  val inFile = sc.textFile("Line_of_numbers.csv")
  val numbersRDD = inFile.map(line => line.split(','))
  scala> numbersRDD.take(10)
  [..]
  14/11/22 12:13:11 INFO SparkContext: Job finished: take at 
  <console>:18, took 0.010062 s
  res7: Array[Array[String]] = Array([42, 42, 55, 61, 53, 49, 43, 47, 49, 60, 68, 54, 34, 35, 35, 39])
  It is an array of String. We need float or double
  val numbersRDD = inFile.map(line => line.split(',')).map(_.toDouble)
  scala> val numbersRDD = inFile.map(line => line.split(',')).map(_.toDouble)
  <console>:15: error: value toDouble is not a member of 
  Array[String]
  val numbersRDD = inFile.map(line => line.split(',')).map(_.toDouble)
  This will not work as we have an array of array of strings. This is where flatMap comes handy!
  scala> val numbersRDD = inFile.flatMap(line => line.split(',')).map(_.toDouble)
  scala> numbersRDD.collect()
  [..]
  res10: Array[Double] = Array(42.0, 42.0, 55.0, 61.0, 53.0, 49.0, 43.0, 47.0, 49.0, 60.0, 68.0, 54.0, 34.0, 35.0, 35.0, 39.0)
  scala> numbersRDD.sum()
  [..]
  14/11/22 12:19:15 INFO SparkContext: Job finished: sum at 
  <console>:18, took 0.013293 s
  res9: Double = 766.0
  scala>
  ```

- **Python:**
  ```python
  inp_file = sc.textFile("Line_of_numbers.csv")
  numbers_rdd = inp_file.map(lambda line: line.split(',')).take(10)
  ```
Loading and Saving Data in Spark

[...]

14/11/22 11:12:25 INFO SparkContext: Job finished: runJob at PythonRDD.scala:300, took 0.023086 s

[[u'42', u'42', u'55', u'61', u'53', u'49', u'47', u'49', u'60', u'68', u'54', u'34', u'35', u'35', u'39']]

>>> But we want the values as integers or double

numbers_rdd = inp_file.flatMap(lambda line: line.split(',')).map(lambda x:float(x))

>>> numbers_rdd.take(10)
14/11/22 11:52:39 INFO SparkContext: Job finished: runJob at PythonRDD.scala:300, took 0.022838 s

[42.0, 42.0, 55.0, 61.0, 53.0, 49.0, 47.0, 49.0, 60.0]

>>> numbers_sum = numbers_rdd.sum()

14/11/22 12:03:16 INFO SparkContext: Job finished: sum at <stdin>:1, took 0.026984 s

>>> numbers_sum

766.0
>>> 

• Java:

import java.util.Arrays;
import java.util.List;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.*;
import org.apache.spark.api.java.function.DoubleFunction;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.SparkFiles;

public class LDSV03 {

    public static void main(String[] args) {
        SparkConf conf = new SparkConf().setAppName("Chapter 05").setMaster("local");
        JavaSparkContext ctx = new JavaSparkContext(conf);
        System.out.println("Running Spark Version : "+ctx.version());
        ctx.addFile("/Users/ksankar/fdps-vii/data/Line_of_numbers.csv");
        //
        JavaRDD<String> lines = ctx.textFile(SparkFiles.get("Line_of_numbers.csv"));
    }
}
//
JavaRDD<String[]> numbersStrRDD = lines.map(new
Function<String,String[]>(){
    public String[] call(String line) {return line.split("","");}
});
List<String[]> val = numbersStrRDD.take(1);
for (String[] e : val) {
    for (String s : e) {
        System.out.print(s+" ");
    }
    System.out.println();
}
//
JavaRDD<String> strFlatRDD = lines.flatMap(new FlatMapFunction
<String,String>() {
    public Iterable<String> call(String line) {return Arrays.
.asList(line.split("",""));}
});
List<String> val1 = strFlatRDD.collect();
for (String s : val1) {
    System.out.print(s+" ");
}
System.out.println();
//
JavaRDD<Integer> numbersRDD = strFlatRDD.map(new
Function<String,Integer>() {
    public Integer call(String s) {return Integer.parseInt(s);}
});
List<Integer> val2 = numbersRDD.collect();
for (Integer s : val2) {
    System.out.print(s+" ");
}
System.out.println();
//
Integer sum = numbersRDD.reduce(new Function2<Integer,Integer,
Integer>() {
    public Integer call(Integer a, Integer b) {return a+b;}
});
System.out.println("Sum = "+sum);
}
Loading and Saving Data in Spark

Running Spark Version : 1.1.1
14/11/22 16:02:18 INFO Utils: Copying /Users/ksankar/Tech/spark/book/Line_of_numbers.csv to /var/folders/gq/70vnyfj6913b61ms_td7gb40000gn/T/spark-9a4bed6d-adb5-4e08-b5c5-5e9089d6e54b/Line_of_numbers.csv
14/11/22 16:02:18 INFO MemoryStore: ensureFreeSpace(163705) called with curMem=0, maxMem=2061647216
14/11/22 16:02:18 INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated size 159.9 KB, free 1966.0 MB)
14/11/22 16:02:18 INFO FileInputFormat: Total input paths to process : 1
14/11/22 16:02:18 INFO SparkContext: Starting job: take at Test03.java:25
[...]
14/11/22 16:02:18 INFO SparkContext: Job finished: take at Test03.java:25, took 0.155961 s
42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
14/11/22 16:02:18 INFO BlockManager: Removing broadcast 1
[...]
14/11/22 16:02:18 INFO SparkContext: Job finished: collect at Test03.java:36, took 0.016938 s
42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
14/11/22 16:02:18 INFO SparkContext: Starting job: collect at Test03.java:45
[...]
14/11/22 16:02:18 INFO SparkContext: Job finished: collect at Test03.java:45, took 0.016657 s
42 42 55 61 53 49 43 47 49 60 68 54 34 35 35 39
14/11/22 16:02:18 INFO SparkContext: Starting job: reduce at Test03.java:51
[...]
14/11/22 16:02:18 INFO SparkContext: Job finished: reduce at Test03.java:51, took 0.019349 s
Sum = 766
This also illustrates one of the ways of getting data out of Spark; you can transform it to a standard Scala array using the `collect()` function. The `collect()` function is especially useful for testing, in much the same way that the `parallelize()` function is. The `collect()` function collects the job's execution results, while `parallelize()` partitions the input data and makes it an RDD. The `collect` function only works if your data fits in memory in a single host (where your code runs on), and even in that case, it adds to the bottleneck that everything has to come back to a single machine.

The `collect()` function brings all the data to the machine that runs the code. So beware of accidentally doing `collect()` on a large RDD!

The `split()` and `toDouble()` functions doesn't always work out so well for more complex CSV files. `opencsv` is a versatile library for Java and Scala. For Python the CSV library does the trick. Let's use the `opencsv` library to parse the CSV files in Scala.

- Scala:

```scala
import au.com.bytecode.opencsv.CSVReader
import java.io.StringReader
sc.addFile("Line_of_numbers.csv")
val inFile = sc.textFile("Line_of_numbers.csv")
val splitLines = inFile.map(line =>
  val reader = new CSVReader(new StringReader(line))
  reader.readNext()
)
val numericData = splitLines.map(line =>
  line.map(_.toDouble)
)
val summedData = numericData.map(row => row.sum)
println(summedData.collect().mkString(",")
```

While loading text files into Spark is certainly easy, text files on local disk are often not the most convenient format for storing large chunks of data. Spark supports loading from all of the different Hadoop formats (sequence files, regular text files, and so on) and from all of the support Hadoop storage sources (HDFS, S3, HBase, and so on). You can also load your CSV into HBase using some of their bulk loading tools (like import TSV) and get your CSV data.
Sequence files are binary flat files consisting of key value pairs; they are one of the common ways of storing data for use with Hadoop. Loading a sequence file into Spark is similar to loading a text file, but you also need to let it know about the types of the keys and values. The types must either be subclasses of Hadoop’s `Writable` class or be implicitly convertible to such a type. For Scala users, some natives are convertible through implicits in `WritableConverter`. As of Version 1.1.0, the standard `WritableConverter` types are `int`, `long`, `double`, `float`, `boolean`, `byte arrays`, and `string`. Let’s illustrate by looking at the process of loading a sequence file of `String` to `Integer`, as shown here:

- **Scala:**
  ```scala
  val data = sc.sequenceFile[String, Int](inputFile)
  ```

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

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

HBase is a Hadoop-based database designed to support random read/write access to entries. Loading data from HBase is a bit different from text files and sequence in files with respect to how we tell Spark what types to use for the data.

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

- Java:

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

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

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

Here are the different ways to save an RDD:

- For Scala:
  ```scala```
  rddOfStrings.saveAsTextFile("out.txt")
  keyValueRdd.saveAsObjectFile("sequenceOut")
  ```

- For Java:
  ```java```
  rddOfStrings.saveAsTextFile("out.txt")
  keyValueRdd.saveAsObjectFile("sequenceOut")
  ```

- For Python:
  ```python```
  rddOfStrings.saveAsTextFile("out.txt")
  ```

In addition, users can save the RDD as a compressed text file by using the following function:

```java```
saveAsTextFile(path: String, codec: Class[_ <: CompressionCodec])
```}

Some references are as follows:

- [http://spark-project.org/docs/latest/scala-programming-guide.html#hadoop-datasets](http://spark-project.org/docs/latest/scala-programming-guide.html#hadoop-datasets)
- [http://commons.apache.org/proper/commons-csv/](http://commons.apache.org/proper/commons-csv/)
Summary

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


Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.