Chapter 5. Loading and Saving Your Data

Both engineers and data scientists will find parts of this chapter useful. Engineers may wish to explore more output formats to see if there is something well suited to their intended downstream consumer. Data scientists can likely focus on the format that their data is already in.

Motivation

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

Spark supports a wide range of input and output sources, partly because it builds on the ecosystem available for Hadoop. In particular, Spark can access data through the `InputFormat` and `OutputFormat` interfaces used by Hadoop MapReduce, which are available for many common file formats and storage systems (e.g., S3, HDFS, Cassandra, HBase, etc.).[5] The section "Hadoop Input and Output Formats" shows how to use these formats directly.

More commonly, though, you will want to use higher-level APIs built on top of these raw interfaces. Luckily, Spark and its ecosystem provide many options here. In this chapter, we will cover three common sets of data sources:

File formats and filesystems

> For data stored in a local or distributed filesystem, such as NFS, HDFS, or Amazon S3, Spark can access a variety of file formats including text, JSON, SequenceFiles, and protocol buffers. We will show how to use several common formats, as well as how to point Spark to different filesystems and configure compression.

Structured data sources through Spark SQL

> The Spark SQL module, covered in Chapter 9, provides a nicer and often more efficient API for structured data sources, including JSON and Apache Hive. We will briefly sketch how to use Spark SQL, but leave the bulk of the details to Chapter 9.

Databases and key/value stores

> We will sketch built-in and third-party libraries for connecting to Cassandra, HBase, Elasticsearch, and JDBC databases.

We chose most of the methods here to be available in all of Spark's languages, but some libraries are still Java and Scala only. We will point out when that is the case.

File Formats

Spark makes it very simple to load and save data in a large number of file formats. Formats range from unstructured, like text, to semistructured, like JSON, to structured, like SequenceFiles (see Table 5-1). The input formats that Spark wraps all transparently handle compressed formats based on the file extension.

Table 5-1. Common supported file formats

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.

Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

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

Text Files

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

LOADING TEXT FILES

Loading a single text file is as simple as calling the `textFile()` function on our SparkContext with the path to the file, as you can see in Examples 5-1 through 5-3. If we want to control the number of partitions we can also specify `minPartitions`.

Example 5-1. Loading a text file in Python

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

Example 5-2. Loading a text file in Scala

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

Example 5-3. Loading a text file in Java

```
JavaRDD<String> input = sc.textFile("file:///home/holden/repos/s
```

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

`wholeTextFiles()` can be very useful when each file represents a certain time period's data. If we had files representing sales data from different periods, we could easily compute the average for each period, as shown in Example 5-4.

Example 5-4. Average value per file in Scala

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

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

SAVING TEXT FILES

Outputting text files is also quite simple. The method `saveAsTextFile()`, demonstrated in Example 5-5, takes a path and will output the contents of the RDD to that file. The path is treated as a directory and Spark will output multiple files underneath that directory. This allows Spark to write the output from multiple nodes. With this method we don't get to control which files end up with which segments of our data, but there are other output formats that do allow this.

Example 5-5. Saving as a text file in Python

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

JSON

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

LOADING JSON

Loading the data as a text file and then parsing the JSON data is an approach that we can use in all of the supported languages. This works assuming that you have one JSON record per row; if you have multiline JSON files, you will instead have to load the whole file and then parse each file. If constructing a JSON parser is expensive in your language, you can use `mapPartitions()` to reuse the parser; see

"Working on a Per-Partition Basis" for details.

There are a wide variety of JSON libraries available for the three languages we are looking at, but for simplicity's sake we are considering only one library per language. In Python we will use the built-in library (http://bit.ly/1upkGQV) (Example 5-6), and in Java and Scala we will use Jackson (http://bit.ly/17k6vli) (Examples 5-7 and 5-8). These libraries have been chosen because they perform reasonably well and are also relatively simple. If you spend a lot of time in the parsing stage, look at other JSON libraries for Scala (http://bit.ly/1xP8JFK) or for Java (http://bit.ly/1upkJu1).

Example 5-6. Loading unstructured JSON in Python

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

In Scala and Java, it is common to load records into a class representing their schemas. At this stage, we may also want to skip invalid records. We show an example of loading records as instances of a Person class.

Example 5-7. Loading JSON in Scala

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

Example 5-8. Loading JSON in Java

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

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

SAVING JSON

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

Let's say we were running a promotion for people who love pandas. We can take our input from the first step and filter it for the people who love pandas, as shown in Examples 5-9 through 5-11.

Example 5-9. Saving JSON in Python

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

Example 5-10. Saving JSON in Scala

```
result.filter(p => P.lovesPandas).map(mapper.writeValueAsString(
  .saveAsTextFile(outputFile)
```

Example 5-11. Saving JSON in Java

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

JavaRDD<Person> result = input.mapPartitions(new ParseJson()).f
```

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

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

Comma-Separated Values and Tab-Separated Values

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

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

LOADING CSV

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

As with JSON, there are many different CSV libraries, but we will use only one for each language. Once again, in Python we use the included `csv` library. In both Scala and Java we use opencsv (http://opencsv.sourceforge.net/).

> **TIP**
>
> There is also a Hadoop InputFormat, `CSVInputFormat` (http://bit.ly/1FigUkq), that we can use to load CSV data in Scala and Java, although it does not support records containing newlines.

If your CSV data happens to not contain newlines in any of the fields, you can load your data with `textFile()` and parse it, as shown in Examples 5-12 through 5-14.

Example 5-12. Loading CSV with textFile() in Python

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

Example 5-13. Loading CSV with textFile() in Scala

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

Example 5-14. Loading CSV with textFile() in Java

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

If there are embedded newlines in fields, we will need to load each file in full and parse the entire segment, as shown in Examples 5-15 through 5-17. This is unfortunate because if each file is large it can introduce bottlenecks in loading and parsing. The different text file loading methods are described "Loading text files".

Example 5-15. Loading CSV in full in Python

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

Example 5-16. Loading CSV in full in Scala

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

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

Example 5-17. Loading CSV in full in Java

```
public static class ParseLine
```

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

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

SAVING CSV

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

The CSV libraries we are using output to files/writers so we can use `StringWriter`/`StringIO` to allow us to put the result in our RDD, as you can see in Examples 5-18 and 5-19.

Example 5-18. Writing CSV in Python

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

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

Example 5-19. Writing CSV in Scala

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

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

SequenceFiles

SequenceFiles are a popular Hadoop format composed of flat files with key/value pairs. SequenceFiles have sync markers that allow Spark to seek to a point in the file and then resynchronize with the record boundaries. This allows Spark to efficiently read SequenceFiles in parallel from multiple nodes. SequenceFiles are a common input/output format for Hadoop MapReduce jobs as well, so if you are working with an existing Hadoop system there is a good chance your data will be available as a SequenceFile.

SequenceFiles consist of elements that implement Hadoop's Writable interface, as Hadoop uses a custom serialization framework. Table 5-2 lists some common types and their corresponding Writable class. The standard rule of thumb is to try adding the word *Writable* to the end of your class name and see if it is a known subclass of `org.apache.hadoop.io.Writable` (http://bit.ly/1FilfEu). If you can't find a Writable for the data you are trying to write out (for example, a custom case class), you can go ahead and implement your own Writable class by overriding `readFields` and `write` from `org.apache.hadoop.io.Writable`.

> **WARNING**
>
> Hadoop's RecordReader reuses the same object for each record, so directly calling `cache` on an RDD you read in like this can fail; instead, add a simple `map()` operation and cache its result. Furthermore, many Hadoop Writable classes do not implement `java.io.Serializable`, so for them to work in RDDs we need to convert them with a `map()` anyway.

Table 5-2. Corresponding Hadoop Writable types

Scala type	Java type	Hadoop Writable
`Int`	`Integer`	`IntWritable` or `VIntWritable`[6]
`Long`	`Long`	`LongWritable` or `VLongWritable`[6]

Float	Float	FloatWritable
Double	Double	DoubleWritable
Boolean	Boolean	BooleanWritable
Array[Byte]	byte[]	BytesWritable
String	String	Text
Array[T]	T[]	ArrayWritable<TW>[7]
List[T]	List<T>	ArrayWritable<TW>[7]
Map[A, B]	Map<A, B>	MapWritable<AW, BW>[7]

In Spark 1.0 and earlier, SequenceFiles were available only in Java and Scala, but Spark 1.1 added the ability to load and save them in Python as well. Note that you will need to use Java and Scala to define custom Writable types, however. The Python Spark API knows only how to convert the basic Writables available in Hadoop to Python, and makes a best effort for other classes based on their available getter methods.

LOADING SEQUENCEFILES

Spark has a specialized API for reading in SequenceFiles. On the SparkContext we can call `sequenceFile(path, keyClass, valueClass, minPartitions)`. As mentioned earlier, SequenceFiles work with Writable classes, so our `keyClass` and `valueClass` will both have to be the correct Writable class. Let's consider loading people and the number of pandas they have seen from a SequenceFile. In this case our `keyClass` would be `Text`, and our `valueClass` would be `IntWritable` or `VIntWritable`, but for simplicity we'll work with `IntWritable` in Examples 5-20 through 5-22.

Example 5-20. Loading a SequenceFile in Python

```
val data = sc.sequenceFile(inFile,
  "org.apache.hadoop.io.Text", "org.apache.hadoop.io.IntWritable
```

Example 5-21. Loading a SequenceFile in Scala

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

Example 5-22. Loading a SequenceFile in Java

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

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

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

SAVING SEQUENCEFILES

Writing the data out to a SequenceFile is fairly similar in Scala. First, because SequenceFiles are key/value pairs, we need a `PairRDD` with types that our SequenceFile can write out. Implicit conversions between Scala types and Hadoop Writables exist for many native types, so if you are writing out a native type you can just save your `PairRDD` by calling `saveAsSequenceFile(path)`, and it will write out the data for you. If there isn't an automatic conversion from our key and value to Writable, or we want to use variable-length types (e.g., `VIntWritable`), we can just map over the data and convert it before saving. Let's consider writing out the data that we loaded in the previous example (people and how many pandas they have seen), as shown in Example 5-23.

Example 5-23. Saving a SequenceFile in Scala

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

In Java saving a SequenceFile is slightly more involved, due to the lack of a `saveAsSequenceFile()` method on the `JavaPairRDD`. Instead, we use Spark's ability to save to custom Hadoop formats, and we will show how to save to a SequenceFile in Java in "Hadoop Input and Output Formats".

Object Files

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

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

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

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

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

Object files are not available in Python, but the Python RDDs and SparkContext support methods called `saveAsPickleFile()` and `pickleFile()` instead. These use Python's `pickle` serialization library. The same caveats for object files apply to pickle files, however: the pickle library can be slow, and old files may not be readable if you change your classes.

Hadoop Input and Output Formats

In addition to the formats Spark has wrappers for, we can also interact with any Hadoop-supported formats. Spark supports both the "old" and "new" Hadoop file

APIs, providing a great amount of flexibility.

LOADING WITH OTHER HADOOP INPUT FORMATS

To read in a file using the new Hadoop API we need to tell Spark a few things. The `newAPIHadoopFile` takes a path, and three classes. The first class is the "format" class, which is the class representing our input format. A similar function, `hadoopFile()`, exists for working with Hadoop input formats implemented with the older API. The next class is the class for our key, and the final class is the class of our value. If we need to specify additional Hadoop configuration properties, we can also pass in a `conf` object.

One of the simplest Hadoop input formats is the `KeyValueTextInputFormat`, which can be used for reading in key/value data from text files (see Example 5-24). Each line is processed individually, with the key and value separated by a tab character. This format ships with Hadoop so we don't have to add any extra dependencies to our project to use it.

Example 5-24. Loading KeyValueTextInputFormat() with old-style API in Scala

```
val input = sc.hadoopFile[Text, Text, KeyValueTextInputFo
  case (x, y) => (x.toString, y.toString)
}
```

We looked at loading JSON data by loading the data as a text file and then parsing it, but we can also load JSON data using a custom Hadoop input format. This example requires setting up some extra bits for compression, so feel free to skip it. Twitter's Elephant Bird package supports a large number of data formats, including JSON, Lucene, Protocol Buffer–related formats, and others. The package also works with both the new and old Hadoop file APIs. To illustrate how to work with the new-style Hadoop APIs from Spark, we'll look at loading LZO-compressed JSON data with `Lzo JsonInputFormat` in Example 5-25.

Example 5-25. Loading LZO-compressed JSON with Elephant Bird in Scala

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

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

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

SAVING WITH HADOOP OUTPUT FORMATS

We already examined SequenceFiles to some extent, but in Java we don't have the same convenience function for saving from a pair RDD. We will use this as a way to illustrate how to use the old Hadoop format APIs (see Example 5-26); the call for the new one (`saveAsNewAPIHadoopFile`) is similar.

Example 5-26. Saving a SequenceFile in Java

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

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

NON-FILESYSTEM DATA SOURCES

In addition to the `hadoopFile()` and `saveAsHadoopFile()` family of functions, you can use `hadoopDataset`/`saveAsHadoopDataSet` and `newAPIHadoopDataset`/`saveAsNewAPIHadoopDataset` to access Hadoop-supported storage formats that are not filesystems. For example, many key/value stores, such as HBase and MongoDB, provide Hadoop input formats that read directly from the key/value store. You can easily use any such format in Spark.

The `hadoopDataset()` family of functions just take a `Configuration` object on which you set the Hadoop properties needed to access your data source. You do the configuration the same way as you would configure a Hadoop MapReduce job, so you can follow the instructions for accessing one of these data sources in MapReduce and then pass the object to Spark. For example, "HBase" shows how to use `newAPIHadoopDataset` to load data from HBase.

EXAMPLE: PROTOCOL BUFFERS

Protocol buffers[9] were first developed at Google for internal remote procedure calls (RPCs) and have since been open sourced. Protocol buffers (PBs) are structured data, with the fields and types of fields being clearly defined. They are optimized to be fast for encoding and decoding and also take up the minimum amount of space. Compared to XML, PBs are 3× to 10× smaller and can be 20× to 100× faster to encode and decode. While a PB has a consistent encoding, there are multiple ways to create a file consisting of many PB messages.

Protocol buffers are defined using a domain-specific language, and then the protocol buffer compiler can be used to generate accessor methods in a variety of languages (including all those supported by Spark). Since PBs aim to take up a minimal amount of space they are not "self-describing," as encoding the description of the data would take up additional space. This means that to parse data that is formatted as PB, we need the protocol buffer definition to make sense of it.

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

PB fields can be many predefined types, or another PB message. These types include `string`, `int32`, `enums`, and more. This is by no means a complete introduction to protocol buffers, so if you are interested you should consult the Protocol Buffers website.

In Example 5-27 we will look at loading many `VenueResponse` objects from a simple protocol buffer format. The sample `VenueResponse` is a simple format with one repeated field, containing another message with required, optional, and enumeration fields.

Example 5-27. Sample protocol buffer definition

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

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

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

Twitter's Elephant Bird library, which we used in the previous section to load JSON data, also supports loading and saving data from protocol buffers. Let's look at writing out some `Venues` in Example 5-28.

Example 5-28. Elephant Bird protocol buffer writeout in Scala

```
val job = new Job()
val conf = job.getConfiguration
LzoProtobufBlockOutputFormat.setClassConf(classOf[Places
val dnaLounge = Places.Venue.newBuilder()
dnaLounge.setId(1);
dnaLounge.setName("DNA Lounge")
dnaLounge.setType(Places.Venue.VenueType.CLUB)
val data = sc.parallelize(List(dnaLounge.build()))
val outputData = data.map{ pb =>
  val protoWritable = ProtobufWritable.newInstance(classOf[
  protoWritable.set(pb)
  (null, protoWritable)
}
outputData.saveAsNewAPIHadoopFile(outputFile, classOf[Text],
```

```
    classOf[ProtobufWritable[Places.Venue]],
    classOf[LzoProtobufBlockOutputFormat[ProtobufWritable
```

A full version of this example is available in the source code for this book.

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

File Compression

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

These compression options apply only to the Hadoop formats that support compression, namely those that are written out to a filesystem. The database Hadoop formats generally do not implement support for compression, or if they have compressed records that is configured in the database itself.

Choosing an output compression codec can have a big impact on future users of the data. With distributed systems such as Spark, we normally try to read our data in from multiple different machines. To make this possible, each worker needs to be able to find the start of a new record. Some compression formats make this impossible, which requires a single node to read in all of the data and thus can easily lead to a bottleneck. Formats that can be easily read from multiple machines are called "splittable." Table 5-3 lists the available compression options.

Table 5-3. Compression options

Format	Splittable	Average compression speed	Effectiveness on text	Hadoc
gzip	N	Fast	High	`org.ap`
lzo	Y	Very fast	Medium	`com.ha`
bzip2	Y	Slow	Very high	`org.ap`
zlib	N	Slow	Medium	`org.ap`
Snappy	N	Very Fast	Low	`org.ap`

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

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

Filesystems

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

Local/"Regular" FS

While Spark supports loading files from the local filesystem, it *requires that the files*

are available at the same path on all nodes in your cluster.

Some network filesystems, like NFS, AFS, and MapR's NFS layer, are exposed to the user as a regular filesystem. If your data is already in one of these systems, then you can use it as an input by just specifying a *file://* path; Spark will handle it as long as the filesystem is mounted at the same path on each node (see Example 5-29).

Example 5-29. Loading a compressed text file from the local filesystem in Scala

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

If your file isn't already on all nodes in the cluster, you can load it locally on the driver without going through Spark and then call `parallelize` to distribute the contents to workers. This approach can be slow, however, so we recommend putting your files in a shared filesystem like HDFS, NFS, or S3.

Amazon S3

Amazon S3 is an increasingly popular option for storing large amounts of data. S3 is especially fast when your compute nodes are located inside of Amazon EC2, but can easily have much worse performance if you have to go over the public Internet.

To access S3 in Spark, you should first set the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables to your S3 credentials. You can create these credentials from the Amazon Web Services console. Then pass a path starting with `s3n://` to Spark's file input methods, of the form `s3n://bucket/path-within-bucket`. As with all the other filesystems, Spark supports wildcard paths for S3, such as `s3n://bucket/my-files/*.txt`.

If you get an S3 access permissions error from Amazon, make sure that the account for which you specified an access key has both "read" and "list" permissions on the bucket. Spark needs to be able to list the objects in the bucket to identify the ones you want to read.

HDFS

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

Using Spark with HDFS is as simple as specifying `hdfs://master:port/path` for your input and output.

> **WARNING**
>
> The HDFS protocol changes across Hadoop versions, so if you run a version of Spark that is compiled for a different version it will fail. By default Spark is built against Hadoop 1.0.4. If you build from source, you can specify `SPARK_HADOOP_VERSION=` as a environment variable to build against a different version; or you can download a different precompiled version of Spark. You can determine the value by running `hadoop version`.

Structured Data with Spark SQL

Spark SQL is a component added in Spark 1.0 that is quickly becoming Spark's preferred way to work with structured and semistructured data. By structured data, we mean data that has a *schema*—that is, a consistent set of fields across data records. Spark SQL supports multiple structured data sources as input, and because it understands their schema, it can efficiently read only the fields you require from these data sources. We will cover Spark SQL in more detail in Chapter 9, but for now, we show how to use it to load data from a few common sources.

In all cases, we give Spark SQL a SQL query to run on the data source (selecting some fields or a function of the fields), and we get back an RDD of `Row` objects, one per record. In Java and Scala, the `Row` objects allow access based on the column number. Each `Row` has a `get()` method that gives back a general type we can cast, and specific `get()` methods for common basic types (e.g., `getFloat()`, `getInt()`, `getLong()`, `getString()`, `getShort()`, and `getBoolean()`). In Python we can just access the elements with `row[column_number]` and `row.column_name`.

Apache Hive

One common structured data source on Hadoop is Apache Hive. Hive can store tables in a variety of formats, from plain text to column-oriented formats, inside HDFS or other storage systems. Spark SQL can load any table supported by Hive.

To connect Spark SQL to an existing Hive installation, you need to provide a Hive configuration. You do so by copying your *hive-site.xml* file to Spark's *./conf/* directory. Once you have done this, you create a `HiveContext` object, which is the entry point to Spark SQL, and you can write Hive Query Language (HQL) queries against your tables to get data back as RDDs of rows. Examples 5-30 through 5-32 demonstrate.

Example 5-30. Creating a HiveContext and selecting data in Python

```
from pyspark.sql import HiveContext

hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT name, age FROM users")
firstRow = rows.first()
print firstRow.name
```

Example 5-31. Creating a HiveContext and selecting data in Scala

```
import org.apache.spark.sql.hive.HiveContext

val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)
val rows = hiveCtx.sql("SELECT name, age FROM users")
```

```
val firstRow = rows.first()
println(firstRow.getString(0)) // Field 0 is the name
```

Example 5-32. Creating a HiveContext and selecting data in Java

```
import org.apache.spark.sql.hive.HiveContext;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SchemaRDD;

HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT name, age FROM users");
Row firstRow = rows.first();
System.out.println(firstRow.getString(0)); // Field 0 is the nam
```

We cover loading data from Hive in more detail in "Apache Hive".

JSON

If you have JSON data with a consistent schema across records, Spark SQL can infer their schema and load this data as rows as well, making it very simple to pull out the fields you need. To load JSON data, first create a `HiveContext` as when using Hive. (No installation of Hive is needed in this case, though—that is, you don't need a *hive-site.xml* file.) Then use the `HiveContext.jsonFile` method to get an RDD of `Row` objects for the whole file. Apart from using the whole `Row` object, you can also register this RDD as a table and select specific fields from it. For example, suppose that we had a JSON file containing tweets in the format shown in Example 5-33, one per line.

Example 5-33. Sample tweets in JSON

```
{"user": {"name": "Holden", "location": "San Francisco"}, "
{"user": {"name": "Matei", "location": "Berkeley"}, "text"
```

We could load this data and select just the username and text fields as shown in Examples 5-34 through 5-36.

Example 5-34. JSON loading with Spark SQL in Python

```
tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

Example 5-35. JSON loading with Spark SQL in Scala

```
val tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
val results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

Example 5-36. JSON loading with Spark SQL in Java

```
SchemaRDD tweets = hiveCtx.jsonFile(jsonFile);
tweets.registerTempTable("tweets");
SchemaRDD results = hiveCtx.sql("SELECT user.name, text FROM twe
```

We discuss more about how to load JSON data with Spark SQL and access its schema in "JSON". In addition, Spark SQL supports quite a bit more than loading data, including querying the data, combining it in more complex ways with RDDs, and running custom functions over it, which we will cover in Chapter 9.

Databases

Spark can access several popular databases using either their Hadoop connectors or custom Spark connectors. In this section, we will show four common connectors.

Java Database Connectivity

Spark can load data from any relational database that supports Java Database Connectivity (JDBC), including MySQL, Postgres, and other systems. To access this data, we construct an `org.apache.spark.rdd.JdbcRDD` and provide it with our SparkContext and the other parameters. Example 5-37 walks you through using `JdbcRDD` for a MySQL database.

Example 5-37. JdbcRDD in Scala

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

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

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

`JdbcRDD` takes several parameters:

- First, we provide a function to establish a connection to our database. This lets each node create its own connection to load data over, after performing any configuration required to connect.
- Next, we provide a query that can read a range of the data, as well as a `lowerBound` and `upperBound` value for the parameter to this query. These parameters allow Spark to query different ranges of the data on different machines, so we don't get bottlenecked trying to load all the data on a single node.[11]
- The last parameter is a function that converts each row of output from a `java.sql.ResultSet` (http://bit.ly/1xSi5DQ) to a format that is useful for manipulating our data. In Example 5-37, we will get `(Int, String)` pairs. If this parameter is left out, Spark will automatically convert each row to an array of objects.

As with other data sources, when using `JdbcRDD`, make sure that your database can

handle the load of parallel reads from Spark. If you'd like to query the data offline rather than the live database, you can always use your database's export feature to export a text file.

Cassandra

Spark's Cassandra support has improved greatly with the introduction of the open source Spark Cassandra connector from DataStax. Since the connector is not currently part of Spark, you will need to add some further dependencies to your build file. Cassandra doesn't yet use Spark SQL, but it returns RDDs of `CassandraRow` objects, which have some of the same methods as Spark SQL's `Row` object, as shown in Examples 5-38 and 5-39.The Spark Cassandra connector is currently only available in Java and Scala.

Example 5-38. sbt requirements for Cassandra connector

```
"com.datastax.spark" %% "spark-cassandra-connector" % "1.0.0-rc5
"com.datastax.spark" %% "spark-cassandra-connector-java" % "1.0.
```

Example 5-39. Maven requirements for Cassandra connector

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

Much like with Elasticsearch, the Cassandra connector reads a job property to determine which cluster to connect to. We set the `spark.cassandra.connection.host` to point to our Cassandra cluster and if we have a username and password we can set them with `spark.cassandra.auth.username` and `spark.cassandra.auth.password`. Assuming you have only a single Cassandra cluster to connect to, we can set this up when we are creating our SparkContext as shown in Examples 5-40 and 5-41.

Example 5-40. Setting the Cassandra property in Scala

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

val sc = new SparkContext(conf)
```

Example 5-41. Setting the Cassandra property in Java

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

The Datastax Cassandra connector uses implicits in Scala to provide some additional functions on top of the SparkContext and RDDs. Let's import the implicit conversions and try loading some data (Example 5-42).

Example 5-42. Loading the entire table as an RDD with key/value data in Scala

```
// Implicits that add functions to the SparkContext & RDDs.
import com.datastax.spark.connector._

// Read entire table as an RDD. Assumes your table test was crea
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);
val data = sc.cassandraTable("test" , "kv")
// Print some basic stats on the value field.
data.map(row => row.getInt("value")).stats()
```

In Java we don't have implicit conversions, so we need to explicitly convert our SparkContext and RDDs for this functionality (Example 5-43).

Example 5-43. Loading the entire table as an RDD with key/value data in Java

```
import com.datastax.spark.connector.CassandraRow;
import static com.datastax.spark.connector.CassandraJavaUtil.

// Read entire table as an RDD. Assumes your table test was crea
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);
JavaRDD<CassandraRow> data = javaFunctions(sc).cassandraTable("t
// Print some basic stats.
System.out.println(data.mapToDouble(new DoubleFunction<Cassandr
  public double call(CassandraRow row) { return row.getInt("
}).stats());
```

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

The Cassandra connector supports saving to Cassandra from a variety of RDD types. We can directly save RDDs of `CassandraRow` objects, which is useful for copying data between tables. We can save RDDs that aren't in row form as tuples and lists by specifying the column mapping, as Example 5-44 shows.

Example 5-44. Saving to Cassandra in Scala

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

This section only briefly introduced the Cassandra connector. For more information, check out the connector's GitHub page.

HBase

Spark can access HBase through its Hadoop input format, implemented in the `org.apache.hadoop.hbase.mapreduce.TableInputFormat` class. This input format returns key/value pairs where the key is of type `org.apache.hadoop.hbase.io.ImmutableBytesWritable` and the value is of

type `org.apache.hadoop.hbase.client.Result`. The `Result` class includes various methods for getting values based on their column family, as described in its API documentation (http://bit.ly/1xSIlPI).

To use Spark with HBase, you can call `SparkContext.newAPIHadoopRDD` with the correct input format, as shown for Scala in Example 5-45.

Example 5-45. Scala example of reading from HBase

```
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.io.ImmutableBytesWrita
import org.apache.hadoop.hbase.mapreduce.TableInputFo

val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "tablename")  // wh

val rdd = sc.newAPIHadoopRDD(
  conf, classOf[TableInputFormat], classOf[ImmutableBytes
```

To optimize reading from HBase, `TableInputFormat` includes multiple settings such as limiting the scan to just one set of columns and limiting the time ranges scanned. You can find these options in the `TableInputFormat` API documentation (http://bit.ly/1xSlz9B) and set them on your `HBaseConfiguration` before passing it to Spark.

Elasticsearch

Spark can both read and write data from Elasticsearch using Elasticsearch-Hadoop. Elasticsearch is a new open source, Lucene-based search system.

The Elasticsearch connector is a bit different than the other connectors we have examined, since it ignores the path information we provide and instead depends on setting up configuration on our SparkContext. The Elasticsearch `OutputFormat` connector also doesn't quite have the types to use Spark's wrappers, so we instead use `saveAsHadoopDataSet`, which means we need to set more properties by hand. Let's look at how to read/write some simple data to Elasticsearch in Examples 5-46 and 5-47.

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

Example 5-46. Elasticsearch output in Scala

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

Example 5-47. Elasticsearch input in Scala

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

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

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

> **WARNING**
>
> On the write side, Elasticsearch can do mapping inference, but this can occasionally infer the types incorrectly, so it can be a good idea to explicitly set a mapping (http://bit.ly/1xSmeaS) if you are storing data types other than strings.

Conclusion

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

5 `InputFormat` and `OutputFormat` are Java APIs used to connect a data source with MapReduce.

6 ints and longs are often stored as a fixed size. Storing the number 12 takes the same amount of space as storing the number 2**30. If you might have a large number of small numbers use the variable sized types, `VIntWritable` and

`VLongWritable`, which will use fewer bits to store smaller numbers.

7 The templated type must also be a Writable type.

8 Hadoop added a new MapReduce API early in its lifetime, but some libraries still use the old one.

9 Sometimes called *pbs* or *protobufs*.

10 Depends on the library used

11 If you don't know how many records there are, you can just do a count query manually first and use its result to determine the `upperBound` and `lowerBound`.

PREV
4. Working with Key/Value Pairs

NEXT
6. Advanced Spark Programming

BOOK SECTION

Installing HBase

from: HBase Essentials by Nishant Garg
Released: November 2014
9 MINS

HBase

BOOK SECTION

HBase in the Hadoop ecosystem

from: Learning HBase by Shashwat Shriparv
Released: November 2014
6 MINS

HBase

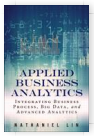
BOOK SECTION

Bibliography

from: Data Structures and Algorithms in Java, 6th Edition by Michael T. Goodrich...
Released: January 2014
10 MINS

Java

BOOK SECTION



Demonstration of Business Analytics Workflows: Analytics Enterprise

from: Applied Business Analytics: Integrating Business Process, Big Data, and Advanced Analytics by Nathaniel Lin
Released: December 2014
18 MINS

Analytics

BOOK SECTION

Reading, Writing, and Using SQL

from: HBase Design Patterns by Mark Kerzner...
Released: December 2014
5 MINS

HBase