Table of Contents for Programming Scala, 2nd Edition

Chapter 18. Scala for Big Data

I said in Chapter 17 that the need to write concurrent programs is driving adoption of FP. However, good concurrency models like actors make it easier for developers to continue using object-oriented programming techniques and avoid the effort of learning functional programming. So, perhaps the *multicore problem* isn't driving change as fast as many of us originally thought?

Now I think that *Big Data* will be a more compelling driver of FP adoption. While actor code still looks object-oriented, more or less, the difference between Big Data applications written in object-oriented Java versus functional Scala is striking. Functional *combinators*, e.g., `map`, `flatMap`, `filter`, `fold`, etc., have always been tools for working with data. Whether that data is in small, in-memory collections or spread across a petabyte-sized cluster, the same abstractions apply. Combinators generalize almost seamlessly across this scale. Once you know the Scala collections, you can pick up the Scala API of one of the popular Big Data tools very quickly. It's true that you'll eventually need to understand how these tools are implemented to write more performant applications, but you'll still be productive quickly.

I've spoken with many Java developers with Big Data experience and little prior interest in Scala. They light up when they see how concise their code could be if they made the switch. For this reason, Scala has emerged as the de facto programming language for Big Data applications, at least among developers like us. *Data Scientists* tend to stick with their favorite tools, such as R and Python.

Big Data: A Brief History

Big Data encompasses the tools and techniques that emerged over the past decade to address three growing challenges. The first was finding ways to work with extremely large data sets, far larger than conventional approaches could manage, like relational databases. The second challenge was the growing need for always-on availability, even when systems experience partial failures.

The early Internet giants like Amazon, eBay, Yahoo!, and Google were the first to face these challenges. They accumulated 100s of TB to several PB (petabytes) of data, far more than could be stored in any relational database and expensive file storage devices, even today. Furthermore, as Internet companies, they needed this data available 24x7. Even the "gold standard" availability of "5-9s" wasn't really good enough. Unfortunately, many early Internet companies experienced embarrassing and catastrophic outages, because they hadn't yet learned how to meet these challenges.

The successful companies attacked the problems from different angles. Amazon, for example, developed a database called *Dynamo* that eschewed the relational model in favor of a simple key-value storage model, with transaction support limited to single rows and sharding of the data around a cluster (described in the famous Dynamo research paper). In exchange, they gained the ability to store much larger data sets by scaling horizontally, with much higher throughput for reads and writes, and with higher availability because node and rack failures would not result in data loss due to replication strategies. Many of the popular *NoSQL* databases today were inspired by Dynamo.

Google developed a clustered, virtualized filesystem, called *Google File System* (GFS), with similar scalability and availability characteristics. On top of GFS, they built a general-purpose computing engine that could distribute an analysis *job* over the cluster, with *tasks* running on many nodes, thereby exploiting parallelism to process the data far more quickly than a single-threaded program could process it. This computing engine, called *MapReduce*,

enabled a wide range of applications to be implemented, from SQL-like queries to machine-learning algorithms.

GFS and MapReduce inspired clone implementations that together came to be called *Hadoop*, which grew rapidly in popularity in the late 2000s as many other companies started using it to store and analyze their own, growing data sets. The file system is called HDFS: Hadoop Distributed File System.

Today, organizations with large data sets often deploy a mixture of Hadoop and NoSQL databases for a range of applications, from reduced-cost data warehousing and other "offline" analysis, to extremely large-scale transaction processing.

"Big Data" is also something of a misnomer because many data sets aren't *that* big, but organizations find the flexibility and low cost of the so-called Big Data tools to be useful for archiving, integrating, and analyzing data in a wide variety of formats and from a wide variety of sources.

For the rest of this chapter, we'll focus on the *computing engine* part of the story, how MapReduce-based tools have evolved and how MapReduce is slowly being replaced by improved successors, with Scala front and center in this evolution.

There are Scala APIs for most of the NoSQL databases, but for the most part, they are conventional APIs similar to Java APIs you might have used. Rather than spend time covering well-understood concepts with Scala veneers, we'll focus instead on more disruptive ideas, using functional programming to simplify and empower data-centric applications.

Improving MapReduce with Scala

The MapReduce Java API is very low level and difficult to use, requiring special expertise to implement nontrivial algorithms and to obtain good performance. The model combines a *map* step, where files are read and the data is converted to key-value pairs as needed for the algorithm. The key-value pairs are shuffled over the cluster to bring together identical keys and then perform final processing in the *reduce* step. Many algorithms require several map-reduce "jobs" sequenced together. Unfortunately, MapReduce flushes data to disk after each job, even when the next job in the sequence will read the data back into memory. The round-trip disk I/O is a major source of inefficiency in MapReduce jobs over massive data sets.

In MapReduce, *map* really means *flat map*, because for every input (say a line from a text file), zero to many output key-value pairs will be generated. *Reduce* has the usual meaning. However, imagine if the Scala containers only had these two combinators, `flatMap` and `reduce`? Many transforms you would like to do would be awkward to implement. Plus, you would need to understand how to do this *efficiently* over large data sets. The upshot is this: while in principle you can implement almost any algorithm in MapReduce, in practice it requires special expertise and challenging programming.

Cascading is the best known Java API that promotes a range of useful abstractions for typical data problems on top of Hadoop MapReduce and it hides many of the low-level MapReduce details (note that as I write this, a replacement backend that eliminates MapReduce is being implemented). Twitter invented *Scalding*, a Scala API on top of Cascading that has become very popular.

Let's examine a classic algorithm, *Word Count*. It is the "Hello World" of Hadoop, because it's conceptually easy to understand, so you can focus on learning an API. In Word Count, a corpus of documents are read in by parallel *map* tasks (usually one task per file). The text is tokenized into words and each map task outputs a sequence of *(word, count)* pairs with the *count* of each *word* found in the document. In the simplest implementation, the mapper just writes *(word, 1)* every time *word* is encountered, but a performance optimization is to only emit one *(word, count)* pair, thereby reducing the number of key-value pairs that get shuffled over the cluster to reducers. The *word* functions as a key in this algorithm.

The shuffling process brings all the same *word* tuples together to the *reducer* tasks, where a final count is done and the tallies are written back to disk. This is why it's logically equivalent to write 10 *(Foo, 1)* tuples or 1 *(Foo, 10)* tuple. Addition is *associative*; it doesn't matter where we add the 10 *Foos*.

Let's compare implementing Word Count in the three APIs: Java MapReduce, Cascading, and Scalding.

Note

Because these examples would require quite a few additional dependencies to build and run, and some of the toolkits don't yet support Scala 2.11, all of the files have the "X" extension so `sbt` doesn't attempt to compile them. Instead, see the footnotes for each example for information on building and using them.

To save space, I'll just show part of the Hadoop MapReduce version. The full source is available in the downloadable code examples for the book at the location shown in the comment:[]

```
// src/main/java/progscala2/bigdata/HadoopWordCount.javaX
...
class WordCountMapper extends MapReduceBase
    implements Mapper<IntWritable, Text, Text, IntWritable> {

  static final IntWritable one  = new IntWritable(1);
  static final Text word = new Text;

  @Override public void map(IntWritable key, Text valueDocContents,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    String[] tokens = valueDocContents.toString.split("\\s+");       //
❶
    for (String wordString: tokens) {
      if (wordString.length > 0) {
        word.set(wordString.toLowerCase);
        output.collect(word, one);
      }
    }
  }
}

class WordCountReduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text keyWord, java.util.Iterator<IntWritable> counts,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    int totalCount = 0;
    while (counts.hasNext) {
//  ❷
      totalCount += counts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```

❶

The actual work of the map step. The text is tokenized and for each *word* found, it is written to the `output` as the key and a count of one is written as the value.

❷

The actual work of the reduce step. For each *word* key, a collection of counts is summed. The word and the final count are written to the output, which will go to files.

This code reminds me of the original EJB 1.X API. Highly invasive and inflexible. Framework boilerplate abounds. You have to wrap all fields in `Writable`, a serialization format, because Hadoop won't do that for you. You have tedious Java idioms. Not shown is the class that provides the `main` routine, which adds another 12 or so lines of setup code. I won't explain all the API details, but hopefully you get the main point that the API requires attention to a lot of details that are unrelated to the trivial algorithm.

Minus import statements and comments, this version is about 60 lines of code. That's not huge. MapReduce jobs tend to be smaller than general-purpose IT applications, but this is also a very simple algorithm. Implementing more sophisticated algorithms raises the complexity level considerably. For example, consider a common SQL query you might write on the output, assuming it's in a table named `wordcount`:

```
SELECT word, count FROM wordcount ORDER BY word ASC, count DESC;
```

Simple. Now search the Internet for *secondary sort mapreduce* and marvel at the complexity of the MapReduce implementations you find.

Cascading provides an intuitive model of *pipes* that are joined into *flows*, where the sources and sinks for data are *taps*. Here is the *full* program (imports elided) for the equivalent Cascading implementation:[]

```
//
src/main/java/progscala2/bigdata/CascadingWordCount.javaX
package impatient;

import ...;

public class CascadingWordCount {
  public static void main( String[] args ) {
    String input  = args[0];
    String output = args[1];

    Properties properties = new Properties();                          // ❶
    AppProps.setApplicationJarClass( properties, Main.class );
    HadoopFlowConnector flowConnector = new HadoopFlowConnector( properties
);

    Tap docTap = new Hfs( new TextDelimited( true, "\t" ), input );    // ❷
    Tap wcTap = new Hfs( new TextDelimited( true, "\t" ), output );

    Fields token = new Fields( "token" );                              // ❸
    Fields text = new Fields( "text" );
    RegexSplitGenerator splitter =
                                       "[ \\[\\]\\
        new RegexSplitGenerator( token, (\\),.]"           );
    Pipe docPipe =                                                     //
❹
      new Each( "token", text, splitter, Fields.RESULTS );

    Pipe wcPipe = new Pipe( "wc", docPipe );                           // ❺
    wcPipe = new GroupBy( wcPipe, token );
    wcPipe = new Every( wcPipe, Fields.ALL, new Count(), Fields.ALL );

    // Connect the taps, pipes, etc., into a
    flow.
    FlowDef flowDef = FlowDef.flowDef()                                //  ❻
     .setName( "wc" )
     .addSource( docPipe, docTap )
     .addTailSink( wcPipe, wcTap );

    // Run the
    flow.
    Flow wcFlow = flowConnector.connect( flowDef );                    // ❼
    wcFlow.complete();
  }
}
```

❶

A small amount of setup code, including configuration for running in Hadoop.

❷

Read and write data using *taps* for HDFS.

❸

Name two *fields* in the tuples representing records. Use a regular expression to split the text into a token stream.

❹

Create a pipe that iterates over the input text and outputs just the words.

❺

Connect a new pipe that performs a group-by operation with the words as the grouping keys. Then append a pipe that counts the sizes of each group.

❻

Create the flow that connects the input and output taps to the pipeline.

❼

Run it.

The Cascading version is about 30 lines without the imports. Even without knowing much about this API, the real algorithm emerges. After we tokenize the corpus into words, we want to *group by* those words and then size each group. That's really all this algorithm does. If we had the "raw words" in a table, the SQL query would be this:

```
SELECT word, COUNT(*) as count FROM raw_words GROUP BY word;
```

Cascading offers an elegant API that has become popular. It is hampered by the relative verbosity of Java and the lack of anonymous functions in pre-Java 8, when the API was created. For example, the `Each`, `GroupBy`, and `Every` objects should be higher-order functions. They are in Scalding.

Here's the Scalding version:[]

```
// src/main/scala/progscala2/bigdata/WordCountScalding.scalaX

import com.twitter.scalding._
// ❶

class WordCount(args : Args) extends Job(args) {

  TextLine(args("input"))
// ❷
    .read
    .flatMap('line -> 'word) {
// ❸
      line: String => line.trim.toLowerCase.split("""\s+""")
    }
    .groupBy('word){ group => group.size('count) }
// ❹
    .write(Tsv(args("output")))
// ❺
}
```

❶

Just one import statement.

❷

Read text files where each line is a "record." `TextLine` abstracts over the local filesystem, HDFS, S3, etc. How you run a Scalding job determines how filesystem paths are interpreted, whereas Cascading requires you to make this choice in the source code. Each line will have the field name `'line`, where Scalding uses Scala symbols to specify fields by name. The expression `args("input")` means grab the *path* from the command-line option `--input path`.

❸

Take each `'line` and tokenize it into words, using `flatMap`. The syntax `('line -> 'word)` means that we select a single input field (there is only one at this point) and the single output field will be called `'word`.

❹

Group by the words and count the group sizes. The output schema of these records will be `('word, 'count)`.

❺

Write the output as tab-separated values to the *path* given on the command line with `--output path`.

The Scalding version is a dozen lines, with the single import statement! Now, almost all the framework details have receded into the background. It's pure algorithm. You *already* know what `flatMap` and `groupBy` are doing, even though Scalding adds an extra argument list to most *combinators* for field selection.

We evolved from a lengthy, tedious *program* to a simple *script*. The whole software development process is changed when you can write such concise programs.

Moving Beyond MapReduce

A growing trend is the need to process events in "real time." MapReduce is only usable for batch-mode jobs. HDFS only recently added support for incremental updates to files. Most Hadoop tools don't support this feature yet.

This trend has led to the creation of new tools, such as *Storm*, a clustered event processing system.

Other growing concerns are the MapReduce performance limitations, such as the excessive disk I/O mentioned previously, and the difficulties of the programming API and underlying model.

Most first-generation technologies have limitations that eventually lead to replacements.The major Hadoop vendors recently embraced a MapReduce replacement called *Spark*, which supports both a batch-mode and streaming model. Spark is written in Scala and it provides excellent performance compared to MapReduce, in part because it caches data in memory between processing steps. Perhaps most important, Spark provides the sort of intuitive API that Scalding provides—incredibly concise, yet expressive.

Where Scalding and Cascading use a pipe metaphor, Spark uses a *Resilient, Distributed Dataset* (RDD), an in-memory data structure distributed over the cluster. It's resilient in the sense that if a node goes down, Spark knows how to reconstruct the missing piece from the source data.

Here is one Spark implementation of Word Count:

```
//
src/main/scala/progscala2/bigdata/WordCountSpark.scalaX
package bigdata

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object SparkWordCount {
  def main(args: Array[String]) = {
                                        "Word
    val sc = new SparkContext("local", Count"      )                  //
❶
    val input = sc.textFile(args(0)).map(_.toLowerCase)               //
❷
    input
      .flatMap(line => line.split("""\W+"""))
// ❸
      .map(word => (word, 1))
// ❹
      .reduceByKey((count1, count2) => count1 + count2)               //
❺
      .saveAsTextFile(args(1))
// ❻
    sc.stop()
// ❼
  }
}
```

❶

Start with a `SparkContext`. The first argument specifies the “master.” In this case, we run locally. The second argument is an arbitrary name for the job.

❷

Load one or more text files from the path specified with the first command-line argument (in Hadoop, directory paths are given and all files in them are read) and convert the strings to lowercase, returning an RDD.

❸

Split on nonalphanumeric sequences of characters, flat-mapping from lines to words.

❹

Map each word to the tuple `(word, 1)`. Recall the output of the Hadoop map tasks for Word Count discussed earlier.

❺

Use `reduceByKey`, which functions like a SQL `GROUP BY` followed by a *reduction*, in this case summing the values in the tuples, the `1`s. In Spark, the first element of a tuple is the default *key* for operations like this and the rest of the tuple is the *value*.

❻

Write the results to the path specified as the second input argument. Spark follows Hadoop conventions and

actually treats the path as a directory to which it writes one "partition" file per final task (with naming convention `part-n`, where `n` is a five-digit number, counting form `00000`).

❼

Shut down the context and stop.

Like the Scalding example, this program is about a dozen lines of code.

Whether you use more mature, but still growing tools like Scalding or up and coming tools like Spark, I hope it's clear that Scala APIs have a unique advantage over Java-based APIs. The functional combinators we already know are the ideal basis for thinking about data analytics, both for users of these tools and also for implementers.

Categories for Mathematics

We discussed *categories* in Chapter 16. A category we didn't discuss that's becoming popular in Big Data is *Monoid*. If you skipped that chapter, just think of *categories* as mathematically oriented *design patterns*.

Monoid is the abstraction for *addition*. It has these properties:

1. A single, associative, binary operation
2. An identity element

Addition of numbers satisfies these properties. We have associativity, *(1.1 + 2.2) + 3.3 == 1.1 + (2.2 + 3.3)*, and *0* is the identity element. Multiplication works, too, with *1* as the identity. Addition and multiplication of numbers are also commutative, *1.1 + 2.2 == 2.2 + 1.1*, but that's *not* required for a Monoid.

What's the big deal? It turns out a *lot* of data structures satisfy these properties, so if you generalize code to work with Monoids, it can be highly reusable (see the list on Wikipedia).

Examples include string concatenation, matrix addition and multiplication, computing maxima and minima, and approximation algorithms like HyperLogLog for finding unique values, Min-hash for set similarity, and Bloom filters for set membership (see Avi Bryant's great "Add ALL The Things!" talk).

Some of these data structures also commute. All can be implemented with parallel execution for high performance over large data sets. The approximation algorithms listed trade off better space efficiency for less accuracy.

You'll see *Monoid* implemented in a number of mathematics packages, including those in the next section.

A List of Scala-Based Data Tools

Besides the Hadoop platform and Scala APIs for databases, a number of tools have emerged for related problems, like general mathematics and Machine Learning. Table 18-1 list some of the active projects you might investigate, including the ones we discussed previously, for completeness.

Table 18-1. Data and mathematics libraries

Option	URL	Description
Algebird	http://bit.ly/10Fk2F7	Twitter's API for abstract algebra that can be used with almost any Big Data API.

Option	URL	Description
Factorie	http://factorie.cs.umass.edu/	A toolkit for deployable *probabilistic modeling*, with a succinct language for creating relational factor graphs, estimating parameters, and performing inference.
Figaro	http://bit.ly/1nWnQf4	A toolkit for *probabilistic programming*.
H2O	http://bit.ly/1G2rfz5	A high-performance, in-memory distributed compute engine for data analytics. Written in Java with Scala and R APIs.
Relate	http://bit.ly/13p17zp	A thin database access layer focused on performance.
ScalaNLP	http://www.scalanlp.org/	A suite of Machine Learning and numerical computing libraries. It is an umbrella project for several libraries, including Breeze, for machine learning and numerical computing, and Epic, for statistical parsing and structured prediction.
ScalaStorm	http://bit.ly/10aaroq	A Scala API for Storm.
Scalding	https://github.com/twitter/scalding	Twitter's Scala API around Cascading that popularized Scala as a language for Hadoop programming.
Scoobi	https://github.com/nicta/scoobi	A Scala abstraction layer on top of MapReduce with an API that's similar to Scalding's and Spark's.
Slick	http://slick.typesafe.com/	A database access layer developed by Typesafe.
Spark	http://spark.apache.org/	The emerging standard for distributed computation in Hadoop environments, as well in Mesos clusters and on single machines ("local" mode).
Spire	https://github.com/non/spire	A numerics library that is intended to be generic, fast, and precise.
Summingbird	https://github.com/twitter/summingbird	Twitter's API that abstracts computation over Scalding (batch mode) and Storm (event streaming).

While the Hadoop environment gets a lot of attention, general-purpose tools like Spark, Scalding/Cascading, and H2O also support smaller deployments, when a large Hadoop cluster is unnecessary.

Recap and What's Next

Few segments of our industry make the case for Scala more strongly than *Big Data*. The way that Scalding and Spark improve upon the Java MapReduce API is striking, even disruptive. Both have made Scala the obvious choice for data-centric application development.

Normally we think of Scala as a statically typed language, like Java. However, the standard library contains a special trait for creating types with more dynamic behavior, like you find in languages such as Ruby and Python, as we'll see in the next chapter. This feature is one tool for building *domain-specific languages* (DSLs), which we'll explore in the chapter after that.