1.6 Counting words with Hadoop—running your first program

Now that you know what the Hadoop and MapReduce framework is about, let's get it running. In this chapter, we'll run Hadoop only on a single machine, which can be your desktop or laptop computer. The next chapter will show you how to run Hadoop over a cluster of machines, which is what you'd want for practical deployment. Running Hadoop on a single machine is mainly useful for development work.

Linux is the official development and production platform for Hadoop, although Windows is a supported development platform as well. For a Windows box, you'll need to install cygwin (http://www-cygwin.com/) to enable shell and Unix scripts.

> **NOTE** Many people have reported success in running Hadoop in development mode on other variants of Unix, such as Solaris and Mac OS X. In fact, MacBook Pro seems to be the laptop of choice among Hadoop developers, as they're ubiquitous in Hadoop conferences and user group meetings.

Running Hadoop requires Java (version 1.6 or higher). Mac users should get it from Apple. You can download the latest JDK for other operating systems from Sun at http://java.sun.com/javase/downloads/index.jsp. Install it and remember the root of the Java installation, which we'll need later.

To install Hadoop, first get the latest stable release at http://hadoop.apache.org/core/releases.html. After you unpack the distribution, edit the script conf/hadoop-env.sh to set JAVA_HOME to the root of the Java installation you have remembered from earlier. For example, in Mac OS X, you'll replace this line

```
# export JAVA_HOME=/usr/lib/j2sdk1.5-sun
```

with this line

```
export JAVA_HOME=/Library/Java/Home
```

You'll be using the Hadoop script quite often. Let's run it without any arguments to see its usage documentation:

```
bin/hadoop
```

We get

```
Usage: hadoop [--config confdir] COMMAND
where COMMAND is one of:
  namenode -format     format the DFS filesystem
  secondarynamenode    run the DFS secondary namenode
  namenode             run the DFS namenode
  datanode             run a DFS datanode
  dfsadmin             run a DFS admin client
  fsck                 run a DFS filesystem checking utility
  fs                   run a generic filesystem user client
  balancer             run a cluster balancing utility
  jobtracker           run the MapReduce job Tracker node
  pipes                run a Pipes job
  tasktracker          run a MapReduce task Tracker node
  job                  manipulate MapReduce jobs
  version              print the version
  jar <jar>            run a jar file
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME <src>* <dest> create a hadoop archive
  daemonlog            get/set the log level for each daemon
 or
  CLASSNAME            run the class named CLASSNAME
Most commands print help when invoked w/o parameters.
```

We'll cover the various Hadoop commands in the course of this book. For our current purpose, we only need to know that the command to run a (Java) Hadoop program is `bin/hadoop jar <jar>`. As the command implies, Hadoop programs written in Java are packaged in jar files for execution.

Fortunately for us, we don't need to write a Hadoop program first; the default installation already has several sample programs we can use. The following command shows what is available in the examples jar file:

```
bin/hadoop jar hadoop-*-examples.jar
```

You'll see about a dozen example programs prepackaged with Hadoop, and one of them is a word counting program called... `wordcount`! The important (inner) classes of that program are shown in listing 1.2. We'll see how this Java program implements the word counting map and reduce functions we had in pseudo-code in listing 1.1. We'll modify this program to understand how to vary its behavior. For now we'll assume it works as expected and only follow the mechanics of executing a Hadoop program.

Without specifying any arguments, executing `wordcount` will show its usage information:

```
bin/hadoop jar hadoop-*-examples.jar wordcount
```

which shows the arguments list:

```
wordcount [-m <maps>] [-r <reduces>] <input> <output>
```

The only parameters are an input directory (`<input>`) of text documents you want to analyze and an output directory (`<output>`) where the program will dump its output. To execute `wordcount`, we need to first create an input directory:

```
mkdir input
```

and put some documents in it. You can add any text document to the directory. For illustration, let's put the text version of the 2002 State of the Union address, obtained from http://www.gpoaccess.gov/sou/. We now analyze its word counts and see the results:

```
bin/hadoop jar hadoop-*-examples.jar wordcount input output
more output/*
```

You'll see a word count of every word used in the document, listed in alphabetical order. This is not bad considering you have not written a single line of code yet! But, also note a number of shortcomings in the included `wordcount` program. Tokenization is based purely on whitespace characters and not punctuation marks, making *States*, *States.*, and *States:* separate words. The same is true for capitalization, where *States* and *states* appear as separate words. Furthermore, we would like to leave out words that show up in the document only once or twice.

Fortunately, the source code for `wordcount` is available and included in the installation at src/examples/org/apache/hadoop/examples/WordCount.java. We can modify it as per our requirements. Let's first set up a directory structure for our playground and make a copy of the program.

```
mkdir playground
mkdir playground/src
mkdir playground/classes
cp src/examples/org/apache/hadoop/examples/WordCount.java
➥ playground/src/WordCount.java
```

Before we make changes to the program, let's go through compiling and executing this new copy in the Hadoop framework.

```
javac -classpath hadoop-*-core.jar -d playground/classes
➥ playground/src/WordCount.java
jar -cvf playground/wordcount.jar -C playground/classes/ .
```

You'll have to remove the output directory each time you run this Hadoop command, because it is created automatically.

```
bin/hadoop jar playground/wordcount.jar
➥ org.apache.hadoop.examples.WordCount input output
```

Look at the files in your output directory again. As we haven't changed any program code, the result should be the same as before. We've only compiled our own copy rather than running the precompiled version.

Now we are ready to modify `WordCount` to add some extra features. Listing 1.2 is a partial view of the WordCount.java program. Comments and supporting code are stripped out.

Listing 1.2 WordCount.java

```
public class WordCount extends Configured implements Tool {

  public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

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

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }

  ...
}
```

1 **Tokenize using white spaces**

2 **Cast token into Text object**

3 **Output count of each token**

The main functional distinction between `WordCount.java` and our MapReduce pseudo-code is that in `WordCount.java`, `map()` processes one line of text at a time whereas our pseudo-code processes a document at a time. This distinction may not even be apparent from looking at `WordCount.java` as it's Hadoop's default configuration.

The code in listing 1.2 is virtually identical to our pseudo-code in listing 1.1 though the Java syntax makes it more verbose. The map and reduce functions are inside inner classes of `WordCount`. You may notice we use special classes such as `LongWritable`, `IntWritable`, and `Text` instead of the more familiar `Long`, `Integer`, and `String` classes of Java. Consider these implementation details for now. The new classes have additional serialization capabilities needed by Hadoop's internal.

The changes we want to make to the program are easy to spot. We see 1 that `WordCount` uses Java's `StringTokenizer` in its default setting, which tokenizes based only on whitespaces. To ignore standard punctuation marks, we add them to the `StringTokenizer`'s list of delimiter characters:

```
StringTokenizer itr = new StringTokenizer(line, " \t\n\r\f,.:;?![]'");
```

When looping through the set of tokens, each token is extracted and cast into a `Text` object (2). (Again, in Hadoop, the special class `Text` is used in place of `String`.) We want the word count to ignore capitalization, so we lowercase all the words before turning them into `Text` objects.

```
word.set(itr.nextToken().toLowerCase());
```

Finally, we want only words that appear more than four times. We modify (3) to collect the word count into the output only if that condition is met. (This is Hadoop's equivalent of the `emit()` function in our pseudo-code.)

```
if (sum > 4) output.collect(key, new IntWritable(sum));
```

After making changes to those three lines, you can recompile the program and execute it again. The results are shown in table 1.1.

Table 1.1 Words with a count higher than 4 in the 2002 State of the Union Address

11th (5)	citizens (9)	its (6)	over (6)	to (123)
a (69)	congress (10)	jobs (11)	own (5)	together (5)
about (5)	corps (6)	join (7)	page (7)	tonight (5)
act (7)	country (10)	know (6)	people (12)	training (5)
afghanistan (10)	destruction (5)	last (6)	protect (5)	united (6)
all (10)	do (6)	lives (6)	regime (5)	us (6)
allies (8)	every (8)	long (5)	regimes (6)	want (5)
also (5)	evil (5)	make (7)	security (19)	war (12)
America (33)	for (27)	many (5)	september (5)	was (11)
American (15)	free (6)	more (11)	so (12)	we (76)
americans (8)	freedom (10)	most (5)	some (6)	we've (5)
an (7)	from (15)	must (18)	states (9)	weapons (12)
and (210)	good (13)	my (13)	tax (7)	were (7)
are (17)	great (8)	nation (11)	terror (13)	while (5)
as (18)	has (12)	need (7)	terrorist (12)	who (18)
ask (5)	have (32)	never (7)	terrorists (10)	will (49)
at (16)	health (5)	new (13)	than (6)	with (22)
be (23)	help (7)	no (7)	that (29)	women (5)
been (8)	home (5)	not (15)	the (184)	work (7)
best (6)	homeland (7)	now (10)	their (17)	workers (5)
budget (7)	hope (5)	of (130)	them (8)	world (17)
but (7)	i (29)	on (32)	these (18)	would (5)
by (13)	if (8)	one (5)	they (12)	yet (8)

Table 1.1 Words with a count higher than 4 in the 2002 State of the Union Address (*continued*)

camps (8)	in (79)	opportunity (5)	this (28)	you (12)
can (7)	is (44)	or (8)	thousands (5)	
children (6)	it (21)	our (78)	time (7)	

We see that 128 words have a frequency count greater than 4. Many of these words appear frequently in almost any English text. For example, there is *a* (69), *and* (210), *i* (29), *in* (79), *the* (184) and many others. We also see words that summarize the issues facing the United States at that time: *terror* (13), *terrorist* (12), *terrorists* (10), *security* (19), *weapons* (12), *destruction* (5), *afghanistan* (10), *freedom* (10), *jobs* (11), *budget* (7), and many others.