

Hadoop in Action

Chuck Lam
James Warren

 MANNING





**MEAP Edition
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Contents

Part I: Hadoop: A distributed programming framework

Chapter 1: Introducing Hadoop

Chapter 2: Starting Hadoop

Chapter 3: Components of Hadoop

Chapter 4: Running Hadoop in the cloud

Part II: Hadoop in action

Chapter 5: Writing programs for distributed processing

Chapter 6: Writing programs for statistical analysis

Chapter 7: Programming practices

Chapter 8: Managing Hadoop

Chapter 9: Cookbook

Part III: Hadoop gone wild

Chapter 10: Hadoop extensions

Chapter 11: Case studies

1

Introducing Hadoop

This chapter covers

- The basics of writing a scalable, distributed data-intensive program
- Understanding Hadoop and MapReduce
- Writing and running a basic Hadoop program

1.1 What is Hadoop?

Hadoop is an open source framework for writing and running distributed applications that process large amounts of data. Distributed computing is a wide and varied field, but the key distinctions of Hadoop are that it is:

- *Accessible*—Hadoop runs on large clusters of commodity machines or on cloud computing services such as Amazon's Elastic Compute Cloud (EC2).
- *Robust*—Because it is intended to run on commodity hardware, Hadoop gracefully handles failures of individual nodes.
- *Scalable*—Hadoop scales linearly as the size of the data increases by adding nodes to the cluster.
- *Simple*—Hadoop allows users to quickly write efficient parallel code.

Because of these features, Hadoop is being widely adopted in both academia and industry.

1.2 Why “Hadoop in Action”?

Speaking from experience, we found our introduction to Hadoop to be tantalizing with its possibilities, yet frustrating to progress beyond coding the basic examples. The documentation at the official Hadoop site is fairly comprehensive, but it wasn't always easy to find straightforward answers to straightforward questions.

We therefore hope to write the book we wish was available when we began our journey using Hadoop. We won't focus on the nitty gritty. Instead we will provide the information that will allow you quickly create useful code, along with answers to problems we encountered going down this road.

1.3 Understanding distributed systems and Hadoop

Moore's law has suited us well for the past decades, but building bigger servers is no longer necessarily the best solution to large scale problems. An alternative that has been gaining in popularity is to use many low-end/commodity machines in conjunction. SETI@home is likely the best known implementation of this approach, as screensavers around the globe are assisting in the search for extra-terrestrial life.

Hadoop subscribes to the approach of using many commodity machines, but it fundamentally differs from most distributed systems in its philosophy towards data. Schemes like SETI@home require a one-time software installation and repeated transmissions of data between clients and servers. But for large scale processing, the size of the data typically dwarfs the size of the processing code. Hadoop thus focuses on moving code to data instead of vice-versa.

As a motivating example consider a high end machine with 4 I/O channels having a throughput of 100 MB/sec. Such a server would require 3 hours simply to read a 4TB dataset! With Hadoop, this dataset would be divided into smaller (typically 64 MB) blocks that are spread amongst the machines in our cluster via the Hadoop Distributed Filesystem (HDFS). With a modest degree of replication, the cluster machines can read the dataset in parallel in a robust manner.

Writing robust parallel code, using distributed filesystems, handling hardware failures – it sounds daunting at first. Don't fret, that's the point of Hadoop – it does the heavy lifting for you. The key point to understand is that Hadoop utilizes both distributed storage and distributed computing, the latter being a framework called MapReduce.

1.4 Understanding MapReduce

You are probably aware of data processing models such as pipelines and messaging queues. These models help programmers in developing different aspects of data processing applications. Pipelines are most familiar to programmers in the form of Unix pipes. Pipelines helps the *reuse* of processing primitives, as new applications are created by the simple chaining of existing ones. Messaging queues help the *synchronization* of processing primitives. In return for the benefits these models provide, the programmer has to (de)compose her data processing task from (into) processing primitives of particular forms. For example, in messaging queues the primitives must behave as either a publisher or a subscriber.

Like pipelines and messaging queues, MapReduce is also a data processing model. Its particular benefit is the easy scaling of data processing over many computing nodes. Under the MapReduce model, the data processing primitives are called *mappers* and *reducers*.

Decomposing a data processing application into mappers and reducers is sometimes non-trivial. However, once an application is written in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is a simple configuration change. This benefit of simple scalability is what has attracted many programmers to the MapReduce model.

Many ways to say MapReduce

Even though much has been written about MapReduce, one does not find the name itself written the same everywhere. The original Google paper and the Wikipedia entry use the CamelCase version "MapReduce." However, Google itself has used "Map Reduce" in some pages on its Web site (for example, <http://research.google.com/roundtable/MR.html>, at least as of this writing). At the official Hadoop documentation site, one can find links pointing to a "Map-Reduce Tutorial." Clicking on the link brings one to a "Hadoop Map/Reduce Tutorial" (http://hadoop.apache.org/core/docs/current/mapred_tutorial.html) explaining the "Map/Reduce" framework. Writing variations also exist for the different Hadoop components such as "NameNode" ("name node", "name-node", and "namenode"), "DataNode", "JobTracker", and "TaskTracker." For the sake of consistency, we will stick to CamelCase for all those terms in this book. (That is, we will use "MapReduce", "NameNode", "DataNode", "JobTracker", and "TaskTracker")

1.4.1 Scaling a simple program manually

Before going through a formal treatment of MapReduce, let's go through an exercise of scaling a simple program to handle a large dataset. By seeing the challenges one encounters, you will better appreciate the benefits of using a framework such as MapReduce to handle the tedious chores for you.

Our exercise is simply to count the number of times each word is in a set of documents. For example, if the set of documents has only one sentence:

Do as I say, not as I do.

Then we want to derive the following word counts:

<u>Word</u>	<u>Count</u>
as	2
do	2
i	2
not	1
say	1

We'll call this particular exercise *word counting*. When the set of documents is small, a straightforward program will do the job. We write one here in pseudo-code:

Listing 1.x. A simple word counting program.

```
define wordCount as Multiset;
for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

The program simply loops through all the documents. For each document, the words are extracted one by one using a tokenization process. For each word, its corresponding entry in a multiset called `wordCount` is incremented by one. At the end, a `display()` function prints out all the entries in `wordCount`.

NOTE

A multiset is a simply a set where each element also has a count. The word count we're trying to come up with is a canonical example of a multiset. In practice it's usually implemented as a hash table.

This program works well enough until the set of documents you want to process gets very large. Say, for example, you plan on building a spam filter so you want to know what words are frequently used in the millions and millions of spam emails you have collected. Looping through all the documents using a single computer will take much longer than you can tolerate. At this point, you rewrite the program so that you can distribute the work over several machines. Each machine will process a distinct fraction of the documents. When all the machines have finished, a second phase of processing will combine the result of all the machines. The pseudo-code for the first phase, to be distributed over many machines, is in listing 1.x. The pseudo-code for the second phase is in listing 1.x.

Listing 1.x. Psuedo-code for the first phase of word counting under distributed processing

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

Listing 1.x. Psuedo-code for the second phase of word counting under distributed processing

```
define totalWordCount as Multiset;
```

```
for each wordCount received from firstPhase {  
    multisetAdd (totalWordCount, wordCount);  
}
```

Now, that didn't seem too hard, right? However, the careful reader will notice a few flaws in our program that may prevent it from working as expected. First of all, we ignore the performance requirement of reading in the documents. If the documents are all stored in one central storage server, then the bottleneck is in the bandwidth of that server. Having more machines for processing only helps up to a certain point until the storage server cannot keep up. However, if the documents are stored in a distributed fashion over the set of processing machines, then each machine can just process the documents that are stored local to it, and the bottleneck of a central storage server is removed. This reiterates the point made earlier about how storage and processing have to be tightly coupled in data-intensive distributed applications.

Another flaw the careful reader will notice is that `wordCount` (and `totalWordCount`) are stored in memory. When processing large document sets, the number of unique words one encounters can exceed the RAM storage of a machine. While it's been reported that the English language has about one million words, a size that fits comfortably into an iPod, our word counting program will have to deal with many unique words not in any standard English dictionaries. For example, we must deal with unique names such as "Hadoop." We have to count misspellings even if they are not real words (for example, "exampel"), and we count all different forms of a word separately (for example, "eat", "ate", "eaten", and "eating"). In any case, even if the number of unique words in the document set is manageable in memory, a slight change to the problem definition can explode the space complexity. For example, instead of words in documents we may be counting IP addresses in a log file. In another example, say we're interested in the frequency of bigrams. In that case, we'll have to work with a multiset with billions of entries, which exceeds the RAM storage of most commodity computers.

NOTE

A bigram is simply a pair of consecutive words. The sentence "Do as I say, not as I do" can be broken into the following bigrams: "Do as", "as I", "I say", "say not", "not as", "as I", "I do." Analogously, trigrams are groups of three consecutive words. Both bigrams and trigrams are important tools in natural language processing.

Since `wordCount` may not fit in memory, we will have to rewrite our program to store this hash table on disk. That is, we'll have to implement a disk-based hash table, which involves a non-trivial amount of coding.

Furthermore, remember that phase two has only one machine, which will have to process `wordCount` sent from *all* the machines in phase one. Just processing one `wordCount` is already quite unwieldy. After we have added enough machines to phase one processing, the one machine in phase two will become a bottleneck. An obvious question to ask is, Can we rewrite phase two in a distributed fashion so that it can scale by adding more machines?

The answer is yes. To make phase two work in a distributed fashion one can divide up its work into multiple machines such that the machines can run independently. One approach to scaling and handling the bulkiness of `wordCount` at the same time is to simply *partition* `wordCount` out such that each machine in phase two only has to handle one partition. For the sake of exposition, let's say we have twenty-six machines for phase two. Each machine will only handle `wordCount` for words beginning with a particular letter in the alphabet. For example, machine 'a' in phase two will only handle word counting for words beginning with the letter 'a'. To enable this partitioning in phase two, a slight modification is needed in phase one. Instead of a single disk-based hash table for `wordCount`, we will want twenty six of them: `wordCount-a`, `wordCount-b`, etc. Each one will only be used for counting words starting with a particular letter. After phase one is finished, `wordCount-a` from each of the machines will be sent to machine 'a' of phase two, all the `wordCount-b`'s will be sent to machine 'b', etc. Each machine in phase one will *shuffle* its results among the machines in phase two.

Looking back, this word counting program is getting complicated. To make it work across a cluster of distributed machines, we found that we need to add a number of functionalities:

- Store files among the processing machines (of phase one).
- Write a disk-based hash table so processing is not limited by RAM capacity.
- Partition the "intermediate data" (that is, `wordCount`) from phase one.
- "Shuffle" the partitions to the appropriate machines in phase two.

Yikes! This is a lot of work for something as simple as word counting, and we haven't even touched upon issues like fault tolerance. (What if a machine fails in the middle of its task?) Hopefully you'll appreciate why one would want a framework like Hadoop. When you write your application in the MapReduce model, Hadoop will take care of all that scalability "plumbing" for you.

1.4.2 Scaling the same program in MapReduce

MapReduce programs are executed in two main phases, called *mapping* and *reducing*. Each phase is defined by a data processing function, and these functions are called *mapper* and *reducer*, respectively. In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper. Afterward, in the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result.

In general, the mapper is meant to *filter and transform* the input into something that the reducer can *aggregate* over. You may see a striking similarity here with the two phases we had to develop in scaling up word counting. The similarity is not accidental. The MapReduce framework was designed after a lot of experience in writing scalable, distributed programs. This two-phase design pattern was seen in scaling many programs, and thus became the basis of the framework.

In scaling our distributed word counting program in the last section, we also had to write the partitioning and shuffling functionalities. It turns out that partitioning and shuffling are

also common design patterns that go along with mapping and reducing. Unlike mapping and reducing though, partitioning and shuffling are generic functionalities that are not very dependent on the particular data processing application. The MapReduce framework thus provide default implementation that would work in most situations.

In order for mapping, reducing, partitioning, and shuffling (and a few others we haven't mentioned) to work together seamlessly, certain structure has to be placed on the data being processed. The data structure, of course, should also be flexible and powerful enough to handle most of the targeted data processing applications. MapReduce uses *lists* and (*key, value*) *pairs* as its main data structures. The keys and values are often integers or strings but can also be dummy values to be ignored or complex object types. The map and reduce functions must obey the following constraint on the types of keys and values.

	Input	Output
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3, v3>)

In the MapReduce framework you write applications by specifying the mapper and reducer. It's instructive to understanding the complete data flow:

1. The input to your application must be structured as a list of (key, value) pairs, `list(<k1, v1>)`. This input format may seem open-ended but is often quite simple in practice. The input format for processing many files is usually just `list(<String filename, String file_content>)`. The input format for processing one big file, such as a log file, is something like `list(<Integer line_number, String log_event>)`.
2. The list of (key, value) pairs is broken up and each individual (key, value) pair, `<k1, v1>`, is processed by calling the map function of the mapper. In practice, the key `k1` is often ignored by mapper. The mapper transforms each `<k1, v1>` pair into a list of `<k2, v2>` pairs. The details of this transformation largely determines what the MapReduce program does. Note that the (key, value) pairs can be processed in arbitrary order, and the transformation must be self contained in that its output is dependent only on one single (key, value) pair.

For word counting, our mapper takes `<String filename, String file_content>` and promptly ignores `filename`. It can output a list of `<String word, Integer count>` but can actually be even simpler. Since we know the counts will be aggregated in a later stage anyways, we can output a list of `<String word, Integer 1>` with repeated entries and let aggregation be done completely later. That is, in the output list we can have the (key, value) pair `<"foo", 3>` once or we can have the pair `<"foo", 1>` three times. As we will see soon, the latter approach is much easier to program. The former approach may have some performance benefits, but let's leave such optimization alone until we have fully grasp the MapReduce framework.

3. The output of all the mappers are (conceptually) aggregated into one giant list of $\langle k2, v2 \rangle$ pairs. All pairs sharing the same $k2$ are grouped together into a new (key, value) pair, $\langle k2, \text{list}(v2) \rangle$. The framework then asks the reducer to process each one of these “aggregated” (key, value) pairs individually. Following our word counting example, the map output for one document may be a list with pair $\langle \text{"foo"}, 1 \rangle$ three times, and the map output for another document may be a list with pair $\langle \text{"foo"}, 1 \rangle$ twice. The “aggregated” pair the reducer will see is $\langle \text{"foo"}, \text{list}(1,1,1,1,1) \rangle$. In word counting, the output of our reducer is $\langle \text{"foo"}, 5 \rangle$, which is the total number of times “foo” has shown up in our document set. Each reducer works on a different word. The MapReduce framework automatically collects all the $\langle k3, v3 \rangle$ pairs and write them to file(s). Note that for the word counting example, the data types $k2$ and $k3$ are the same and $v2$ and $v3$ are also the same. This will not always be the case for other data processing applications.

Let's rewrite the word counting program in MapReduce to see how this all fit together.

Listing 1.x. Pseudo-code for map and reduce functions for word counting

```
map(String filename, String document) {
    List<String> T = tokenize(document);
    for each token in T {
        emit ((String)token, (Integer) 1);
    }
}

reduce(String token, List<Integer> values) {
    Integer sum = 0;
    for each value in values {
        sum = sum + value;
    }
    emit ((String)token, (Integer) sum);
}
```

Previously we have said that the output of both map and reduce function are lists. As you can see from the pseudo-code, in practice we use a special function in the framework called `emit()` to generate the elements in the list one at a time. This `emit()` function further relieves the programmer from managing what can be a very large list.

The code looks very much like what we have in listing 1.x, except this time it will actually work at scale. MapReduce does make building scalable distributed programs look easy, does it?

1.5 Counting words with Hadoop—running your first program

Now that you know what the Hadoop and MapReduce framework is about, let's go ahead and get it running. In this chapter, we will 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 the set up you would want for practical deployment. However, running Hadoop on a single machine will still be useful for development work.

Linux is the official development and production platform for Hadoop, although Windows is supported for development as well. For a Windows box, you will 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. These other operating systems are not officially “supported” mostly in the sense of documentation.

Running Hadoop requires Java (version 1.6 or higher). You can download the latest JDK from Sun at <http://java.sun.com/javase/downloads/index.jsp>. Install it and remember the root of the Java installation, which we will need later.

To install Hadoop, you 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` so that `JAVA_HOME` is set to the root of the Java installation you have remembered from earlier. For example, in Mac OS X, you will replace this line:

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

with this line:

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

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

```
bin/hadoop
```

We get

```
Usage: hadoop [--config confdir] COMMAND
```

where COMMAND is one of:

<code>namenode -format</code>	format the DFS filesystem
<code>secondarynamenode</code>	run the DFS secondary namenode
<code>namenode</code>	run the DFS namenode
<code>datanode</code>	run a DFS datanode
<code>dfsadmin</code>	run a DFS admin client
<code>fsck</code>	run a DFS filesystem checking utility
<code>fs</code>	run a generic filesystem user client
<code>balancer</code>	run a cluster balancing utility
<code>jobtracker</code>	run the MapReduce job Tracker node
<code>pipes</code>	run a Pipes job
<code>tasktracker</code>	run a MapReduce task Tracker node
<code>job</code>	manipulate MapReduce jobs
<code>version</code>	print the version
<code>jar <jar></code>	run a jar file
<code>distcp <srcurl> <desturl></code>	copy file or directories recursively
<code>archive -archiveName NAME <src>* <dest></code>	create a hadoop archive
<code>daemonlog</code>	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 will cover the various Hadoop commands throughout this book. For our current purpose, we just 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.

Luckily 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 should 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.x. In a moment, we will explain how this Java program implements the word counting map and reduce functions we had in pseudo-code in listing 1.x, and we will also explain how you can modify this program to vary its behavior. For now we will assume it works as expected and we just want to show 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. So to execute `wordcount`, we need to first create an input directory

```
mkdir input
```

and put some documents in it. You can put any text documents there. For illustration, we will put the text version of the 2002 State of the Union address, obtained from <http://www.gpoaccess.gov/sou/>, into the input directory. We now analyze its word counts and see the results:

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

You will 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! However, you will also notice a number of shortcomings in the included `wordcount` program. Tokenization is based purely on whitespace characters and not punctuation marks, so "States", "States.", and "States:" are all considered separate words. We would also like to ignore capitalization, so "States" and "states" can be counted together. Furthermore, most words have a small count, showing up in the document only once or twice, and we don't really care about those infrequent words.

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 to do what we want. We 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
```

[CA]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
```

[CA]playground/src/WordCount.java

```
jar -cvf playground/wordcount.jar -C playground/classes/ .
```

You will have to remove the output directory each time you run this Hadoop command, since it will attempt to create that directory.

```
bin/hadoop jar playground/wordcount.jar
```

[CA]org.apache.hadoop.examples.WordCount input output

Look at the files in your output directory again. Since we haven't changed any program code, the result should be the same as before. We just went through compiling our own copy rather than running their pre-compiled version.

Now we are ready to modify WordCount to add some extra features. Listing 1.x is a partial view of the WordCount.java program. Comments and supporting code are stripped out.

Listing 1.x 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));
        }
    }
}
```

```
} ...
```

Replace #1, #2, and #3 in the following paragraph with cueballs

The main functional distinction between `WordCount.java` and our MapReduce pseudo-code is that `WordCount.java` is set up such that `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 simply a use of Hadoop's default configuration.

Otherwise, the code in listing 1.x is virtually identical to our pseudo-code in listing 1.x. Of course, the Java syntax makes it a bit 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. It's really an implementation detail for Hadoop as these new classes provide serialization capabilities better tuned for Hadoop.

The changes we want to make to the program are easy to spot. We see in line #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 in line #2. (In Hadoop, the special class `Text` is used in place of `String`.) We want word count to ignore capitalization so we lower case all the words before turning them into `Text` objects.

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

Finally, we only care about words with counts higher than four. We therefore 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.x.

Table 1.x. Words that have occurred more than four times 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)
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)	

It appears that 128 words have a frequency count greater than four. 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. However, we also see many 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.

1.2 History of Hadoop

Hadoop started out as a subproject of Nutch, which in turn was a subproject of Apache Lucene. Doug Cutting founded all three projects in his spare time, and each project was a logical progression of the previous one.

Lucene is a full-featured text indexing and searching library. Given a text collection, a developer can easily add search capability to the documents using the Lucene engine. Desktop search, enterprise search, and many domain-specific search engines have been built using Lucene. In particular, Nutch is the most ambitious extension of Lucene, as it tries to build a complete Web search engine using Lucene as its core component. Nutch has parsers for HTML, a Web crawler, a link-graph database, and other extra components necessary for a Web search engine. Doug Cutting envisions Nutch to be an open democratic alternative to the proprietary technologies in commercial offerings such as Google.

Besides having extra components like crawler and parser, a Web search engine differs from a basic document search engine in terms of scale. Whereas Lucene is targeted at indexing millions of documents, Nutch should be able to handle billions of Web pages without becoming exorbitantly expensive to operate. Nutch will have to run on a distributed cluster of commodity hardware. The challenge for the Nutch team is to address scalability issues in software. Nutch needs a layer to handle distributed processing, redundancy, automatic failover, and load balancing. These challenges are by no means trivial.

Fortunately, around 2004, Google published two papers describing the Google File System (GFS) and the MapReduce framework. Google claimed to use these two technologies for scaling its own search system. Doug Cutting immediately saw the applicability of these technologies to Nutch, and his team implemented the new framework and ported Nutch to it. The new implementation gave an immediate boost to Nutch's scalability. It started to handle several hundred million Web pages and could run on clusters of dozens of nodes. Doug realized that a dedicated project to flesh out the two technologies was needed to get to Web scale, and Hadoop was born. Yahoo! hired Doug in January 2006 to work with a dedicated team on improving Hadoop as an open source project. Two years later Hadoop achieved the status of an Apache Top Level Project. Later, in February 19, 2008, Yahoo! announced that Hadoop running on a 10,000+ core Linux cluster was its production system for indexing the Web (<http://developer.yahoo.net/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html>). Finally, Hadoop has truly hit Web scale.

What's up with the names?

When naming software projects, Doug Cutting seems to have gotten much inspiration from his family. "Lucene" is his wife's middle name, and her maternal grandmother's first name. His son, as a toddler, used "Nutch" as the all-purpose word for "meal" and later gave the name "Hadoop" to a yellow stuffed elephant. When thinking of a name for Hadoop, Doug said he "was looking for a name that wasn't already a web domain and

wasn't trademarked, so I tried various words that were in my life but not used by anybody else. Kids are pretty good at making up words."

1.6 Controversy around MapReduce

While MapReduce has been heavily promoted by its proponents, it is not without its detractors. In fact, no less an authority than Michael Stonebraker has pointed out some weaknesses of MapReduce from a database perspective. (According to Wikipedia, Stonebraker's "career covers, and helped create, the majority of the existing relational database market today. He is also the founder of Ingres, Illustra, Cohera, StreamBase Systems and Vertica and was previously the CTO of Informix.") In a blog post titled "MapReduce: A major step backwards," co-authored with David DeWitt, Stonebraker criticizes MapReduce as (quote):

4. A giant step backward in the programming paradigm for large-scale data intensive applications
5. A sub-optimal implementation, in that it uses brute force instead of indexing
6. Not novel at all—it represents a specific implementation of well known techniques developed nearly 25 years ago
7. Missing most of the features that are routinely included in current DBMS
8. Incompatible with all of the tools DBMS users have come to depend on

Hadoop, being an open source version of MapReduce, allows any competent developer to fix bugs and make improvements. So even as Stonebraker and DeWitt were writing about the shortcomings of MapReduce, some people were starting to address them. For example, no less than three separate projects (Pig, Hive, and Jaql) had sprung up to provide higher-level access languages on top of Hadoop, one of the more important shortcomings Stonebraker and DeWitt had pointed out. The lack of optimized execution from using techniques such as indexing were being discussed in different guises within the Hadoop community. In fact, in this book we will show a few techniques for you to write MapReduce programs that are more optimized than simple brute force approach.

1.7 Summary

Hadoop is a versatile tool that quickly allows new users to access the power of distributed computing. By using distributed storage and transferring code instead of data, Hadoop avoids the costly transmission step when working with large datasets. Moreover, the redundancy of data allows Hadoop to gracefully recover should a single node fail.

We demonstrated the ease of creating programs with Hadoop using the MapReduce framework. What is just as important is what you didn't have to do – worry about partitioning the data, determining which nodes will perform which tasks, or handling messages between nodes. Since Hadoop handles this for you, you can focus on what's most important to you – your data and what you want to do with it.

In the next chapter we'll go into further details about setting up Hadoop. The "out-of-the-box" configuration is fine for introductory examples, but Hadoop is extremely flexible and easily customized to suit your needs.

1.8 Further reading

The official Hadoop Web site is at

- <http://hadoop.apache.org/>

The original papers on the Google File System and MapReduce are well worth reading, especially to get an appreciation of their underlying design and architecture:

- The Google File System - <http://labs.google.com/papers/gfs.html>
- MapReduce: Simplified Data Processing on Large Clusters - <http://labs.google.com/papers/mapreduce.html>

Michael Stonebraker and David DeWitt's criticism of MapReduce, along with readers' commentary, are in two blog posts:

- <http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html>
- <http://www.databasecolumn.com/2008/01/mapreduce-continued.html>