Hadoop in Action

Chuck Lam

MEAP Unedited Draft

MANNING

MEAP Edition
Manning Early Access Program

TABLE OF CONTENTS

PART1 Hadoop - A Distributed Programming Framework

PART 2 - Hadoop in Action

PART 3 - Hadoop Gone Wild

Part 1

Hadoop–A Distributed Programming Framework

Part 1 of this book introduces the basics for understanding and using Hadoop. We describe the hardware components that make up a Hadoop cluster, as well as the installation and configuration to create a working system. We cover the MapReduce framework at a high level and get your first MapReduce program up and running.

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 MapReduce program

Today, we're surrounded by data. People upload videos, take pictures on their cell phones, text friends, update their Facebook status, leave comments around the web, click on ads, and so forth. Machines, too, are generating and keeping more and more data. You may even be reading this book as digital data on your computer screen, and certainly your purchase of this book is recorded as data with some retailer.[1]

The exponential growth of data first presented challenges to cutting-edge businesses such as Google, Yahoo, Amazon, and Microsoft. They needed to go through terabytes and petabytes of data to figure out which websites were popular, what books were in demand, and what kinds of ads appealed to people. Existing tools were becoming inadequate to process such large data sets. Google was the first to publicize *MapReduce*—a system they had used to scale their data processing needs.

---

[1] Of course, you're reading a legitimate copy of this, right?

This system aroused a lot of interest because many other businesses were facing similar scaling challenges, and it wasn't feasible for everyone to reinvent their own proprietary tool. Doug Cutting saw an opportunity and led the charge to develop an open source version of this MapReduce system called Hadoop. Soon after, Yahoo and others rallied around to support this effort. Today, Hadoop is a core part of the computing infrastructure for many web companies, such as Yahoo, Facebook, LinkedIn, and Twitter. Many more traditional businesses, such as media and telecom, are beginning to adopt this system too. Our case studies in chapter 12 will describe how companies including New York Times, China Mobile, and IBM are using Hadoop.

Hadoop, and large-scale distributed data processing in general, is rapidly becoming an important skill set for many programmers. An effective programmer, today, must have knowledge of relational databases, networking, and security, all of which were considered optional skills a couple decades ago. Similarly, basic understanding of distributed data processing will soon become an essential part of every programmer's toolbox. Leading universities, such as Stanford and CMU, have already started introducing Hadoop into their computer science curriculum. This book will help you, the practicing programmer, get up to speed on Hadoop quickly and start using it to process your data sets.

This chapter introduces Hadoop more formally, positioning it in terms of distributed systems and data processing systems. It gives an overview of the MapReduce programming model. A simple word counting example with existing tools highlights the challenges around processing data at large scale. You'll implement that example using Hadoop to gain a deeper appreciation of Hadoop's simplicity. We'll also discuss the history of Hadoop and some perspectives on the MapReduce paradigm. But let me first briefly explain why I wrote this book and why it's useful to you.

1.1 Why "Hadoop in Action"?

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

The purpose of writing the book is to address this problem. I won't focus on the nitty-gritty details. Instead I will provide the information that will allow you to quickly create useful code, along with more advanced topics most often encountered in practice.

1.2 What is Hadoop?

Formally speaking, 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).

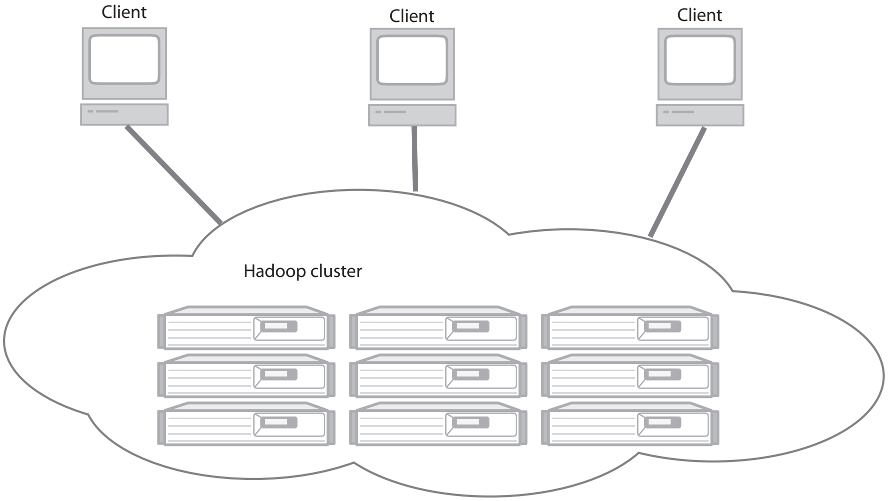

Figure 1.1 A Hadoop cluster has many parallel machines that store and process large data sets. Client computers send jobs into this computer cloud and obtain results.

- *Robust*—Because it is intended to run on commodity hardware, Hadoop is architected with the assumption of frequent hardware malfunctions. It can gracefully handle most such failures.
- *Scalable*—Hadoop scales linearly to handle larger data by adding more nodes to the cluster.
- *Simple*—Hadoop allows users to quickly write efficient parallel code.

Hadoop's accessibility and simplicity give it an edge over writing and running large distributed programs. Even college students can quickly and cheaply create their own Hadoop cluster. On the other hand, its robustness and scalability make it suitable for even the most demanding jobs at Yahoo and Facebook. These features make Hadoop popular in both academia and industry.

Figure 1.1 illustrates how one interacts with a Hadoop cluster. As you can see, a Hadoop cluster is a set of commodity machines networked together in one location.[2] Data storage and processing all occur within this "cloud" of machines. Different users can submit computing "jobs" to Hadoop from individual clients, which can be their own desktop machines in remote locations from the Hadoop cluster.

Not all distributed systems are set up as shown in figure 1.1. A brief introduction to other distributed systems will better showcase the design philosophy behind Hadoop.

---

[2] While not strictly necessary, machines in a Hadoop cluster are usually relatively homogeneous x86 Linux boxes. And they're almost always located in the same data center, often in the same set of racks.

1.3 Understanding distributed systems and Hadoop

Moore's law suited us well for the past decades, but building bigger and bigger servers is no longer necessarily the best solution to large-scale problems. An alternative that has gained popularity is to tie together many low-end/commodity machines together as a single functional *distributed system*.

To understand the popularity of distributed systems (scale-out) vis-à-vis huge monolithic servers (scale-up), consider the price performance of current I/O technology. A high-end machine with four I/O channels each having a throughput of 100 MB/sec will require three hours to *read* a 4 TB data set! With Hadoop, this same data set will be divided into smaller (typically 64 MB) blocks that are spread among many machines in the cluster via the Hadoop Distributed File System (HDFS). With a modest degree of replication, the cluster machines can read the data set in parallel and provide a much higher throughput. And such a cluster of commodity machines turns out to be cheaper than one high-end server!

The preceding explanation showcases the efficacy of Hadoop relative to monolithic systems. Now let's compare Hadoop to other architectures for distributed systems. SETI@home, where screensavers around the globe assist in the search for extraterrestrial life, represents one well-known approach. In SETI@home, a central server stores radio signals from space and serves them out over the internet to client desktop machines to look for anomalous signs. This approach moves the data to where computation will take place (the desktop screensavers). After the computation, the resulting data is moved back for storage.

Hadoop differs from schemes such as SETI@home in its philosophy toward data. SETI@home requires repeat transmissions of data between clients and servers. This works fine for computationally intensive work, but for data-intensive processing, the size of data becomes too large to be moved around easily. Hadoop focuses on moving code to data instead of vice versa. Referring to figure 1.1 again, we see both the data and the computation exist within the Hadoop cluster. The clients send only the MapReduce programs to be executed, and these programs are usually small (often in kilobytes). More importantly, the move-code-to-data philosophy applies within the Hadoop cluster itself. Data is broken up and distributed across the cluster, and as much as possible, computation on a piece of data takes place on the same machine where that piece of data resides.

This move-code-to-data philosophy makes sense for the type of data-intensive processing Hadoop is designed for. The programs to run ("code") are orders of magnitude smaller than the data and are easier to move around. Also, it takes more time to move data across a network than to apply the computation to it. Let the data remain where it is and move the executable code to its hosting machine.

Now that you know how Hadoop fits into the design of distributed systems, let's see how it compares to data processing systems, which usually means SQL databases.

1.4 Comparing SQL databases and Hadoop

Given that Hadoop is a framework for processing data, what makes it better than standard relational databases, the workhorse of data processing in most of today's applications? One reason is that SQL (*structured* query language) is by design targeted at structured data. Many of Hadoop's initial applications deal with unstructured data such as text. From this perspective Hadoop provides a more general paradigm than SQL.

For working only with structured data, the comparison is more nuanced. In principle, SQL and Hadoop can be complementary, as SQL is a query language which can be implemented on top of Hadoop as the execution engine.[3] But in practice, SQL databases tend to refer to a whole set of legacy technologies, with several dominant vendors, optimized for a historical set of applications. Many of these existing commercial databases are a mismatch to the requirements that Hadoop targets.

With that in mind, let's make a more detailed comparison of Hadoop with typical SQL databases on specific dimensions.

SCALE-OUT INSTEAD OF SCALE-UP

Scaling commercial relational databases is expensive. Their design is more friendly to scaling up. To run a bigger database you need to buy a bigger machine. In fact, it's not unusual to see server vendors market their expensive high-end machines as "database-class servers." Unfortunately, at some point there won't be a big enough machine available for the larger data sets. More importantly, the high-end machines are not cost effective for many applications. For example, a machine with four times the power of a standard PC costs a lot more than putting four such PCs in a cluster. Hadoop is designed to be a scale-out architecture operating on a cluster of commodity PC machines. Adding more resources means adding more machines to the Hadoop cluster. Hadoop clusters with ten to hundreds of machines is standard. In fact, other than for development purposes, there's no reason to run Hadoop on a single server.

KEY/VALUE PAIRS INSTEAD OF RELATIONAL TABLES

A fundamental tenet of relational databases is that data resides in tables having relational structure defined by a schema. Although the relational model has great formal properties, many modern applications deal with data types that don't fit well into this model. Text documents, images, and XML files are popular examples. Also, large data sets are often unstructured or semistructured. Hadoop uses key/value pairs as its basic data unit, which is flexible enough to work with the less-structured data types. In Hadoop, data can originate in any form, but it eventually transforms into (key/value) pairs for the processing functions to work on.

FUNCTIONAL PROGRAMMING (MAPREDUCE) INSTEAD OF DECLARATIVE QUERIES (SQL)

SQL is fundamentally a high-level declarative language. You query data by stating the result you want and let the database engine figure out how to derive it. Under MapReduce you

---

[3] This is in fact a hot area within the Hadoop community, and we'll cover some of the leading projects in chapter 11.

specify the actual steps in processing the data, which is more analogous to an execution plan for a SQL engine. Under SQL you have query statements; under MapReduce you have scripts and codes. MapReduce allows you to process data in a more general fashion than SQL queries. For example, you can build complex statistical models from your data or reformat your image data. SQL is not well designed for such tasks.

On the other hand, when working with data that do fit well into relational structures, some people may find MapReduce less natural to use. Those who are accustomed to the SQL paradigm may find it challenging to think in the MapReduce way. I hope the exercises and the examples in this book will help make MapReduce programming more intuitive. But note that many extensions are available to allow one to take advantage of the scalability of Hadoop while programming in more familiar paradigms. In fact, some enable you to write queries in a SQL-like language, and your query is automatically compiled into MapReduce code for execution. We'll cover some of these tools in chapters 10 and 11.

OFFLINE BATCH PROCESSING INSTEAD OF ONLINE TRANSACTIONS

Hadoop is designed for offline processing and analysis of large-scale data. It doesn't work for random reading and writing of a few records, which is the type of load for online transaction processing. In fact, as of this writing (and in the foreseeable future), Hadoop is best used as a write-once, read-many-times type of data store. In this aspect it's similar to data warehouses in the SQL world.

You have seen how Hadoop relates to distributed systems and SQL databases at a high level. Let's learn how to program in it. For that, we need to understand Hadoop's MapReduce paradigm.

1.5 Understanding MapReduce

You're probably aware of data processing models such as pipelines and message queues. These models provide specific capabilities in developing different aspects of data processing applications. The most familiar pipelines are the Unix pipes. Pipelines can help the *reuse* of processing primitives; simple chaining of existing modules creates new ones. Message queues can help the *synchronization* of processing primitives. The programmer writes her data processing task as processing primitives in the form of either a producer or a consumer. The timing of their execution is managed by the system.

Similarly, MapReduce is also a data processing model. Its greatest advantage is the easy scaling of data processing over multiple 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 nontrivial. But, once you write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This 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 website (for example, http://research.google.com/roundtable/MR.html). 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*, *namenode*, and *namenode*), *DataNode*, *JobTracker*, and *TaskTracker*. For the sake of consistency, we'll go with CamelCase for all those terms in this book. (That is, we will use *MapReduce*, *NameNode*, *DataNode*, *JobTracker*, and *TaskTracker*.)

1.5.1 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 process a large data set. You'll see the challenges of scaling a data processing program and will better appreciate the benefits of using a framework such as MapReduce to handle the tedious chores for you.

Our exercise is to count the number of times each word occurs in a set of documents. In this example, we have a set of documents having only one document with only one sentence:

> *Do as I say, not as I do.*

We derive the word counts shown to the right.

Word	Count
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. Let's write one here in pseudo-code:

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

The program 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 set where each element also has a count. The word count we're trying to generate is a canonical example of a multiset. In practice, it's usually implemented as a hash table.

This program works fine until the set of documents you want to process becomes large. For example, you want to build a spam filter to know the words frequently used in the millions of spam emails you've received. Looping through all the documents using a single computer will be extremely time consuming. You speed it up by rewriting the program so that it distributes the work over several machines. Each machine will process a distinct fraction of the documents. When all the machines have completed this, 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

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

And the pseudo-code for the second phase is

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

That wasn't too hard, right? But a few details 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 can't keep up. You'll also need to split up the documents among the set of processing machines such that each machine will process only those documents that are stored in it. This will remove the bottleneck of a central storage server. This reiterates the point made earlier about storage and processing having to be tightly coupled in data-intensive distributed applications.

Another flaw with the program is that `wordCount` (and `totalWordCount`) are stored in memory. When processing large document sets, the number of unique words can exceed the RAM storage of a machine. The English language has about one million words, a size that fits comfortably into an iPod, but our word counting program will deal with many unique words not found in any standard English dictionary. 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*). Even if the number of unique words in the document set is manageable in memory, a slight change in the problem definition can explode the space complexity. For example, instead of words in documents, we may want to count IP addresses in a log file, or the frequency of bigrams. In the case of the latter, we'll work with a multiset with billions of entries, which exceeds the RAM storage of most commodity computers.

> **NOTE** A bigram is 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 in natural language processing.

`wordCount` may not fit in memory; we'll have to rewrite our program to store this hash table on disk. This means we'll implement a disk-based hash table, which involves a substantial amount of coding.

Furthermore, remember that phase two has only one machine, which will process `wordCount` sent from *all* the machines in phase one. Processing one `wordCount` is itself quite unwieldy. After we have added enough machines to phase one processing, the single machine in phase two will become the bottleneck. The obvious question 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, you must somehow divide its work among multiple machines such that they can run independently. You need to *partition* `wordCount` after phase one such that each machine in phase two only has to handle one partition. In one example, let's say we have 26 machines for phase two. We assign each machine to 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, we need a slight modification in phase one. Instead of a single disk-based hash table for `wordCount`, we will need 26 of them: `wordCount-a`, `wordCount-b`, and so on. Each one counts words starting with a particular letter. After phase one, `wordCount-a` from each of the phase one machines will be sent to machine A of phase two, all the `wordCount-b`'s will be sent to machine B, and so on. 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 find that we need to add a number of functionalities:

- Store files over many processing machines (of phase one).
- Write a disk-based hash table permitting processing without being limited by RAM capacity.
- Partition the intermediate data (that is, `wordCount`) from phase one.
- Shuffle the partitions to the appropriate machines in phase two.

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?) This is the reason why you 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.

1.5.2 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. In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result.

In simple terms, 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 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 functions. Partitioning and shuffling are common design patterns that go along with mapping and reducing. Unlike mapping and reducing, though, partitioning and shuffling are generic functionalities that are not too dependent on the particular data processing application. The MapReduce framework provides a default implementation that works in most situations.

In order for mapping, reducing, partitioning, and shuffling (and a few others we haven't mentioned) to seamlessly work together, we need to agree on a common structure for the data being processed. It should 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 primitives. 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. Let's look at 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 multiple files is usually `list(<String filename, String file_content>)`. The input format for processing one large file, such as a log file, is `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 the mapper. The mapper transforms each `<k1, v1>` pair into a list of `<k2, v2>` pairs. The details of this transformation largely determine what the MapReduce program does. Note that the (key/value) pairs are processed in arbitrary order. 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 be even simpler. As we know the counts will be aggregated in a later stage, we can output a list of `<String word, Integer 1>` with repeated entries and let the complete aggregation be done 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'll see, 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 grasped the MapReduce framework.
3. The output of all the mappers are (conceptually) aggregated into one giant list of `<k2, v2>` pairs. All pairs sharing the same `k2` are grouped together into a new (key/value) pair, `<k2, list(v2)>`. The framework 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 `<"foo", 1>` three times, and the map output for another document may be a list with pair `<"foo", 1>` twice. The aggregated pair the reducer will see is `<"foo", list(1,1,1,1,1)>`. In word counting, the output of our reducer is `<"foo", 5>`, which is the total number of times "foo" has occurred in our document set. Each reducer works on a different word. The MapReduce framework automatically collects all the <k3, v3> pairs and writes 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 all this fits together Listing 1.1 shows the pseudo-code.

Listing 1.1 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);
}
```

We've said before 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 a large list.

The code looks similar to what we have in section 1.5.1, except this time it will actually work at scale. Hadoop makes building scalable distributed programs easy, doesn't it? Now let's turn this pseudo-code into a Hadoop program.

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);                  ①
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());                                      ②
        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));                        ③
    }
  }

  ...
}
```

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 ❷. (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 ❸ 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)
a (69)
about (5)
act (7)
afghanistan (10)
all (10)
allies (8)
also (5)
America (33)
American (15)
americans (8)
an (7)
and (210)
are (17)
as (18)
ask (5)
at (16)
be (23)
been (8)
best (6)
budget (7)
but (7)
by (13)

citizens (9)
congress (10)
corps (6)
country (10)
destruction (5)
do (6)
every (8)
evil (5)
for (27)
free (6)
freedom (10)
from (15)
good (13)
great (8)
has (12)
have (32)
health (5)
help (7)
home (5)
homeland (7)
hope (5)
i (29)
if (8)

its (6)
jobs (11)
join (7)
know (6)
last (6)
lives (6)
long (5)
make (7)
many (5)
more (11)
most (5)
must (18)
my (13)
nation (11)
need (7)
never (7)
new (13)
no (7)
not (15)
now (10)
of (130)
on (32)
one (5)

over (6)
own (5)
page (7)
people (12)
protect (5)
regime (5)
regimes (6)
security (19)
september (5)
so (12)
some (6)
states (9)
tax (7)
terror (13)
terrorist (12)
terrorists (10)
than (6)
that (29)
the (184)
their (17)
them (8)
these (18)
they (12)

to (123)
together (5)
tonight (5)
training (5)
united (6)
us (6)
want (5)
war (12)
was (11)
we (76)
we've (5)
weapons (12)
were (7)
while (5)
who (18)
will (49)
with (22)
women (5)
work (7)
workers (5)
world (17)
would (5)
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.

1.7 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, 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. Nutch is the most ambitious extension of Lucene. 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 added components like a crawler and a 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.

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 immediately boosted 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, on 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). Hadoop had truly hit web scale!

What's up with the names?

When naming software projects, Doug Cutting seems to have been inspired by 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 named a yellow stuffed elephant *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."

1.8 Summary

Hadoop is a versatile tool that 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 data sets. Moreover, the redundancy of data allows Hadoop to recover should a single node fail. You have seen the ease of creating programs with Hadoop using the MapReduce framework. What is equally important is what you didn't have to do—worry about partitioning the data, determining which nodes will perform which tasks, or handling communication between nodes. Hadoop handles this for you, leaving you free to 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 the internals of Hadoop and setting up a working Hadoop cluster.

1.9 Resources

The official Hadoop website is at http://hadoop.apache.org/.

The original papers on the Google File System and MapReduce are well worth reading. Appreciate 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