

# Hadoop

## IN ACTION

SECOND EDITION

Chuck P. Lam  
Mark W. Davis



 MANNING



**MEAP Edition  
Manning Early Access Program  
Hadoop in Action, Second Edition  
Version 6**

Copyright 2015 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# Welcome

---

Dear Reader,

In the four years after the publication of *Hadoop in Action*, interest in and development of the platform has not just continued, but thrived. Today a record number of organizations are using Hadoop to meet complex data analysis challenges while the open source community has improved the efficiency and reliability of the platform for a wide range of use cases. Simultaneously, a new job title has emerged: data scientist. Tasked with creating value out of data assets, these new pioneers are leveraging the complete information picture of an organization to drive repeatable and measurable business objectives.

In *Hadoop in Action, 2nd Edition*, we have deeply revised the original book to cover all of the supporting technologies that make up the modern Hadoop “ecosystem.” We cover the architectural changes that impact MapReduce programming, YARN applications, Tez optimizations, and how to use these systems with emerging best practices. We also start at the very beginning and show why Hadoop is important and how to get started. Later we delve into advanced topics in data management and data science, including machine learning and predictive analytics that drive automated decision-making.

We hope you have as much fun reading and trying the examples as we have had in writing the book. Hadoop remains a fascinating addition to the information economy and we think its impact will continue for many years.

Best,  
— Mark Davis and Chuck Lam

# *brief contents*

---

## **PART 1: HADOOP—A DISTRIBUTED PROGRAMMING FRAMEWORK**

- 1 Introducing Hadoop*
- 2 Starting a Hadoop*
- 3 Securing the Hadoop platform*
- 4 Components of Hadoop*

## **PART 2: HADOOP IN ACTION**

- 5 Writing basic MapReduce programs*
- 6 Advanced MapReduce*
- 7 Programming practices*

## **PART 3: DATA MANAGEMENT WITH HADOOP**

- 8 Data Security for Data Management*
- 9 SQL meets Hadoop with Hive*
- 10 Programming with Pig*
- 11 HBase: The Hadoop Analytics Database*

## **PART 4: HADOOP FOR DATA SCIENTISTS & CYBER-SECURITY ANALYTICS**

- 12 MapReduce programming for data science*
- 13 Writing and Using YARN Applications*
- 14 Collaborative filtering using Spark*
- 15 Accelerating SQL analytics with Tez*
- 16 Machine Learning on Hadoop*
- 17 Data Exfiltration Analysis for Cyber Security*

## **APPENDIXES:**

- A HDFS file commands*
- B HiveQL*
- C Pig Latin*
- D HBase Commands*

# *Part 1*

## *Hadoop—A Distributed Programming Framework*

Part 1 of this book introduces the basics for understanding and using Hadoop. We start with an overview of distributed data processing and how Hadoop enhances and improves on other ways of managing data. We then cover the MapReduce framework at a high level and get your first MapReduce program up and running. Finally, we describe all of the different software components that are part of Hadoop, as well as many complementary technologies, and then show how you can use them to create data workflows.

## 1

# *Introducing Hadoop*

## ***This chapter covers***

- The history and origins of Hadoop and Big Data technologies
- 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.<sup>1</sup>

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.

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

---

<sup>1</sup> Of course, you're reading a legitimate copy of this, right?

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 X 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, a 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 we wrote this book and why it's useful to you.

## 1.1 Why “Hadoop in Action”?

Speaking from experience, we 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. We won't focus on the nitty-gritty details. Instead, we will provide the information that will allow you to quickly create useful code, along with more advanced topics most often encountered in practice. The 2<sup>nd</sup> Edition of this book expands on the original in important ways, too. We decided to not just change the existing material but to expand on the examples for the rapidly changing topic of data science. Hence, you will find all new sections that cover advanced topics related to understanding and processing data, as well as a practical description of how to integrate Hadoop into data management systems.

## 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).
- *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.<sup>2</sup>

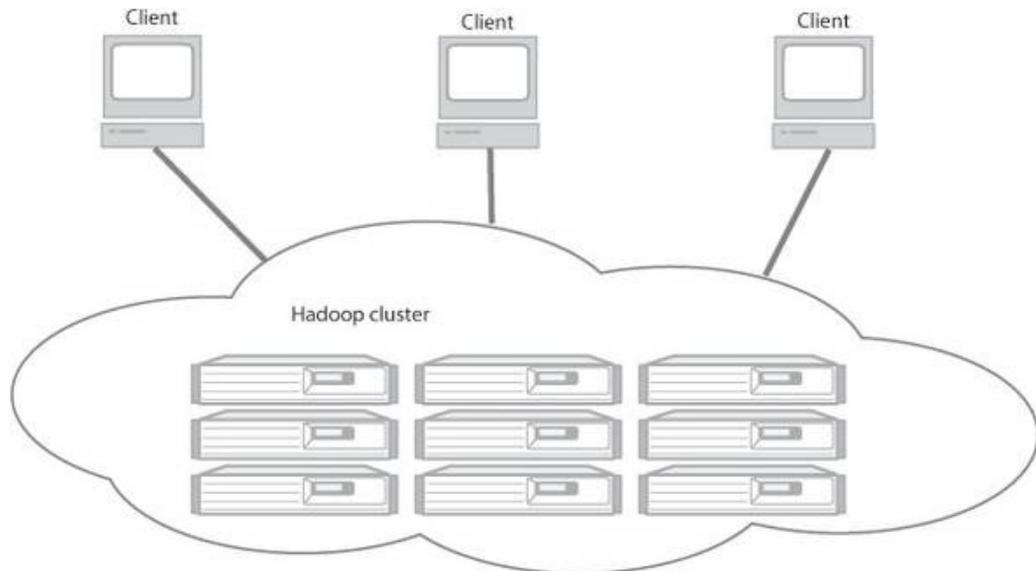


Figure 1.1 A Hadoop cluster has many parallel machines that store and process large data sets. Client computers send jobs into this computer cloud and obtain results.

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.

---

<sup>2</sup> 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.

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.

### **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, but also includes so-called NoSQL databases that rely on different approaches to data storage.

## **1.4 Comparing Hadoop with SQL and NoSQL databases**

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.<sup>3</sup> 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.

### **1.4.1 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.

### **1.4.2 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.

---

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

### **1.4.3 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 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 later chapters.

### **1.4.4 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.

### **1.4.5 NoSQL versus SQL**

NoSQL databases are databases that don't use SQL as their access patterns. These databases are fairly new and growing in popularity, although there have been many types of non-relational databases through the last several decades (e.g., ISAM, object, XML, search). There are several reasons why these new NoSQL databases were created, and many of them parallel the reasons given above for Hadoop as a general alternative to SQL data processing. For example, key-value stores dominate many of the NoSQL approaches. The key-value store is simple for many applications, and can be designed to be inexpensive and fault-tolerant. A second kind of NoSQL database is based on column representations of data rather than tabular ones like most SQL storage systems. These "analytics databases" can use SQL extensions or even completely new APIs for accessing the data. A popular one, HBase, even runs on Hadoop, using Hadoop as its data storage layer.

You have seen how Hadoop relates to distributed systems, SQL and NoSQL 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 MapReduce 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](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, as well as the Hadoop 1.X components like 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
as	2

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 a smartphone, 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*). There are also numbers and addresses that are often unique. 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, and making it efficient for this task can be very complicated.

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 fact, in the original work at Google, the task was to create search indexes that contain vectors of document URLs for each word in the web; the pages were tokenized and then the combined lists aggregated together, much like the word counter presented here.

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.

	<b>input</b>	<b>output</b>
map	<code>&lt;k1, v1&gt;</code>	<code>list(&lt;k2, v2&gt;)</code>
reduce	<code>&lt;k2, list(v2)&gt;</code>	<code>list(&lt;k3, v3&gt;)</code>

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 (for instance, it may be the line number of the incoming text in the value). 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  $\langle k_2, v_2 \rangle$  pairs. All pairs sharing the same  $k_2$  are grouped together into a new (key/value) pair,  $\langle k_2, \text{list}(v_2) \rangle$ . 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  $\langle \text{"foo"}, 1 \rangle$  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 occurred in our document set. Each reducer works on a different word. The MapReduce framework automatically collects all the  $\langle k_3, v_3 \rangle$  pairs and writes them to file(s). Note that for the word-counting example, the data types  $k_2$  and  $k_3$  are the same and  $v_2$  and  $v_3$  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 being supported by the Hadoop community with the help of Microsoft in recent releases.

**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 is the minimum recommended; later versions are better). 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/releases.html> (scroll down and select the Download link). After you unpack the distribution, edit the script `etc/hadoop/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:
    fs                run a generic filesystem user client
    version           print the version
    jar <jar>         run a jar file
    checknative [-a|-h] check native hadoop and compression libraries availability
    distcp <srcurl> <desturl> copy file or directories recursively
    archive -archiveName NAME -p <parent path> <src>* <dest> create a hadoop archive
    classpath         prints the class path needed to get the
                     Hadoop jar and the required libraries
    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. You can find the examples jar file in the `share/hadoop/mapreduce` directory. The following command shows what is available in the examples jar file:

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

which shows the arguments list:

```
Usage: wordcount <in> <out>
```

The only parameters are an input directory (`<in>`) of text documents you want to analyze and an output directory (`<out>`) 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/> (select the More Information links to get the text version). We now analyze its word counts and see the results:

```
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-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, packaged into a jar file in the `share` subdirectory hierarchy. Unjar the source package directly into the top level of your Hadoop distribution:

```
jar -xf share/hadoop/mapreduce/sources/hadoop-mapreduce-examples-2.4.1-sources.jar
```

The source for `wordcount` is then at `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 org/apache/hadoop/examples/WordCount.java playground/src/
```

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

```
javac -classpath share/hadoop/common/lib/commons-cli-
1.2.jar:share/hadoop/common/hadoop-common-
2.4.1.jar:share/hadoop/mapreduce/hadoop-mapreduce-client-core-
2.4.1.jar:share/hadoop/mapreduce/lib/commons-io-
2.4.jar:share/hadoop/common/lib/hadoop-annotations-2.4.1.jar
playground/src/WordCount.java -d playground/classes

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

Note the complicated classpath required for compilation. Hadoop 2.0 segregated functional jars into subdirectories and is now compiled using Maven. This means that the classpaths for doing simple things from the command line get ugly. 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 {

    public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{

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

        public void map(Object key, Text value, Context context
            ) throws IOException, InterruptedException {
```

```

StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
}
}
}

public static class IntSumReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context
            ) throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
...
}

```

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 `IntWritable`, and `Text` instead of the more familiar `Integer` and `String` classes of Java. Consider these implementation details for now. The new classes have additional serialization capabilities needed by Hadoop's internal engine.

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.

sdasdsad

**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)
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 was originally conceived of as 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 envisioned Nutch to be an open democratic alternative to the proprietary technologies in commercial offerings such as Google. As Nutch evolved, other search servers like Apache Solr became popular and provided a scalable framework around Lucene for search. Nutch then returned to its roots as primarily a crawler that can then post its crawled documents to Solr for indexing.

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 The Hadoop Ecosystem**

The phenomenal growth of the core Hadoop system has resulted in the development of other projects and tools to make working with Hadoop easier. The result is what can be considered an ecosystem around Hadoop. Some of these software components were originally intended for other, more general purposes, but were fitted to Hadoop when it was realized that they added value to the system. In other cases, solutions to Hadoop needs graduated to becoming standalone technologies. It can be a bit daunting to try to find exactly the right tool for the job in the Hadoop ecosystem; there is often more than one answer to a question of how to get a job done.

In this section, we provide an overview of important flora and fauna in the Hadoop ecosystem, and some suggestions to how they can play a part in solving data management and analysis problems.

### **1.8.1 Apache Zookeeper**

Apache Zookeeper is a coordination service that is used by Hadoop to manage the state across the entire cluster. As a service, Zookeeper accepts the creation of named items that can then be watched by other processes in the cluster. So, for example, when a MapReduce job starts on one node, that state is recorded to Zookeeper and the task trackers can wait until the state changes at the Zookeeper before transmitting results to Reducers. Zookeeper services work as a group and require a quorum before continuing, supporting robust responses when one or more nodes fail.

Zookeeper is both part of Hadoop and is also a standalone system that can be used by other distributed processing technologies, and is a part of systems like Spark (see below).

### **1.8.2 YARN: Yet Another Resource Negotiator**

YARN is a new addition to Hadoop with Hadoop 2.0. YARN abstracts process management for Hadoop away from just MapReduce execution workflows while providing resource and utilization management for the cluster as a whole. In prior versions of Hadoop, a JobTracker staged the MapReduce code to the data and TaskTrackers executed the code on that node. With YARN, abstract containers can run MapReduce code or can run other processes as well. As we will see later, you can even write a YARN application to perform a unique task on Hadoop, though many of these applications have already been written to do common tasks and support other technologies.

Some new YARN-specific applications include:

- **Spark:** In-memory distributed data processing that breaks problems up over all of the Hadoop nodes, but keeps the data in memory for better performance. Key to Spark is the idea of a Resilient Distributed Dataset (RDD) that contains the details needed to rebuild the dataset from an external store (usually Hadoop HDFS) if the Spark node or process dies.
- **Tez:** Tez is a new execution engine that works with YARN to improve the performance of Hadoop data processing tasks. Previously, when SQL-like queries using HiveQL were

processed, an execution plan was created that would distribute the MapReduce jobs to fulfill the query. Each of the data processing tools had its own workflow engine and they were not fully optimized to take advantage of YARN. Now, the workflow engine for all of these data processing tools can be replaced with Tez that provides integrated YARN scheduling and container optimization. In reality, you will generally not interact with Tez, but the tools you do will have Tez as an option.

- Giraph: A common data processing challenge with big data is finding relationships between people, products, emails, musical tastes, and so forth. Conceptually, these relationships can be seen as a massive graph with nodes representing things and lines representing relationships. Using a graph like that, it becomes easy to ask who I communicate with who also likes the same kind of music that I like. Enter Giraph, a graph processing framework on top of Hadoop and now YARN. Giraph does large-scale in-memory processing for graphs to avoid having to write them out on Hadoop HDFS and iteratively reprocess them.
- MLBase: With Hadoop to analyze data, a next step was to add machine learning to the suite of analysis tools. For instance, Yahoo! uses regression methods to try to predict what kinds of advertising is more likely to get clicked on by users given their past behavior and the behaviors of others. A project called Mahout (a Sanskrit word for an elephant tamer) was developed to provide MapReduce-enabled machine learning algorithms. With the movement towards YARN and in-memory data processing, work began at UC Berkeley to create a library that could provide the core algorithms needed for machine learning, but could do so over Spark. MLBase, still in early development, provides high-performance machine learning.

### **1.8.3 Hive**

Hive is a SQL execution engine that runs on Hadoop. Hive doesn't support full SQL standards, however; it offers a subset instead with notable gaps in join execution and other areas. Hive began around the same time as Pig but with the clear intent to provide a bridge for data professionals who had SQL training. Hive treats tabular data on HDFS as if it were a database table. Using Data Definition Language (DDL) syntax, you can create a schema for the data on the Hadoop filesystem. The schema and tabular metadata that are created are, somewhat ironically, stored in a persistent metastore that is often a relational database. New technologies related to Hive include Tez (described above) that enhances query performance on YARN, and HCatalog that provides common metastore functions for Hive, Pig, and other applications.

### **1.8.4 Oozie**

As developers and data scientists began moving MapReduce jobs from development and into production, there was a need for a way to specify workflows of different MapReduce jobs. In order to create workflows, there also had to be the ability to monitor jobs and job failures, to invoke additional steps, and generally to control Hadoop. Oozie was born from those needs

and provides a workflow orchestrator that can execute and monitor complex sequences of Hadoop operations.

### **1.8.5 Avro**

Avro fills an important need in the Hadoop data processing ecosystem, that of data serialization. Avro creates data containers that have schema associated with them and that can be serialized into binary and then deserialized very efficiently.

### **1.8.6 HBase**

Another NoSQL technology that is built into Hadoop is HBase. HBase stores data in tables but stores any number of columns in each row. This is very different from RDBMS tables that store a defined number of columns in a row to create a rectangular data matrix. HBase was designed to mimic Google's BigTable distributed data store, just like HDFS mimics Google File System (GFS). HBase supports querying data based on a key combined with a timestamp and column identifier. So, for instance, I might create a table that contains all of word counts for a given page on the web combined into one column, and then also have a column that contains just the top five word occurrences, and another that might contain a category for the page ("might" because the page may not have been processed yet). Another row can be created for the same page at a later time with both instances differing by their timestamps. In any case, HBase is very high performance for doing many lookups that can be fulfilled with key, timestamp, and column, as well as many scans for aggregated lookups.

### **1.8.7 Pig**

Pig is a data processing engine that uses a specialized language called PigLatin to specify schema and data manipulation. It can be used for querying like Hive, but it also can be used to transform data on Hadoop. PigLatin has looping primitives and schema assignment functions, and can output results back to HDFS. Pig operates on data by sequencing together MapReduce programs that do small, atomic tasks, much like Hive and HiveQL. And like Hive, Pig can be run using Tez as its workflow engine when operating in a YARN Hadoop environment.

### **1.8.8 Flume**

Getting data into Hadoop is the first step to processing and analyzing the data. Flume is a system for moving data into HDFS from remote systems using configurable memory-resident daemons that watch for data on those systems and then forward the data to Hadoop. Flume is used to transfer web server log files into Hadoop for analysis, for example, but can also be applied to other data producers and customized for other needs using a data flow language.

### **1.8.9 Solr**

Solr is a search server built around Lucene. We already mentioned Lucene in the discussion of the history of Hadoop, but what is notable about Solr is that it is now integrated back into Hadoop and can store its indexes on HDFS. Further work is ongoing to improve other aspects

of Solr performance on Hadoop, but it serves an important addition to the data processing capabilities of Hadoop by providing text search for unstructured data sources on Hadoop. Indexing text into Solr requires converting the data into an XML format that Solr can ingest and that reflects a searchable schema for the data.

### **1.8.10 Impala**

Cloudera, the Hadoop vendor, developed an accelerator for Hive called Impala that they sell commercially but have also licensed under Apache open-source terms. Impala adds query planning to each data node, and queries can coordinate between nodes. This emulates the way many commercial parallel data warehouses operate and provides significant speedups for certain types of queries.

### **1.8.11 Sqoop**

Sqoop follows the same model as Flume, above, but is specifically targeted at moving data from databases to Hadoop for processing there. A common scenario that is discussed in the next section is combining online transaction processing systems with web traffic to understand customer behaviors. Sqoop can pull data from the transactional database to make it available on Hadoop.

## **1.9 Big Data Workflows**

The motivation for Hadoop at Yahoo! was to support user log file processing. Doing that at scale for hundreds of millions of customers made possible things like newsfeed tuning. Later, using Hadoop email spam filtering was enhanced to more rapidly detect and build models for new, devious email spamming approaches. In both cases, these began as simple projects but required transferring data from file systems and databases to Hadoop, running increasingly complex analytic workflows over that data, and then populating dashboard, building models, or providing visualizations to help understand the data and its impact.

We now see Hadoop as providing end-to-end data processing capabilities that typically enhance database-driven enterprises (rather than replace the databases). In this section, we will take a look at an example big data workflow typical for a web retailer. Figure 1 shows the data architecture for this example

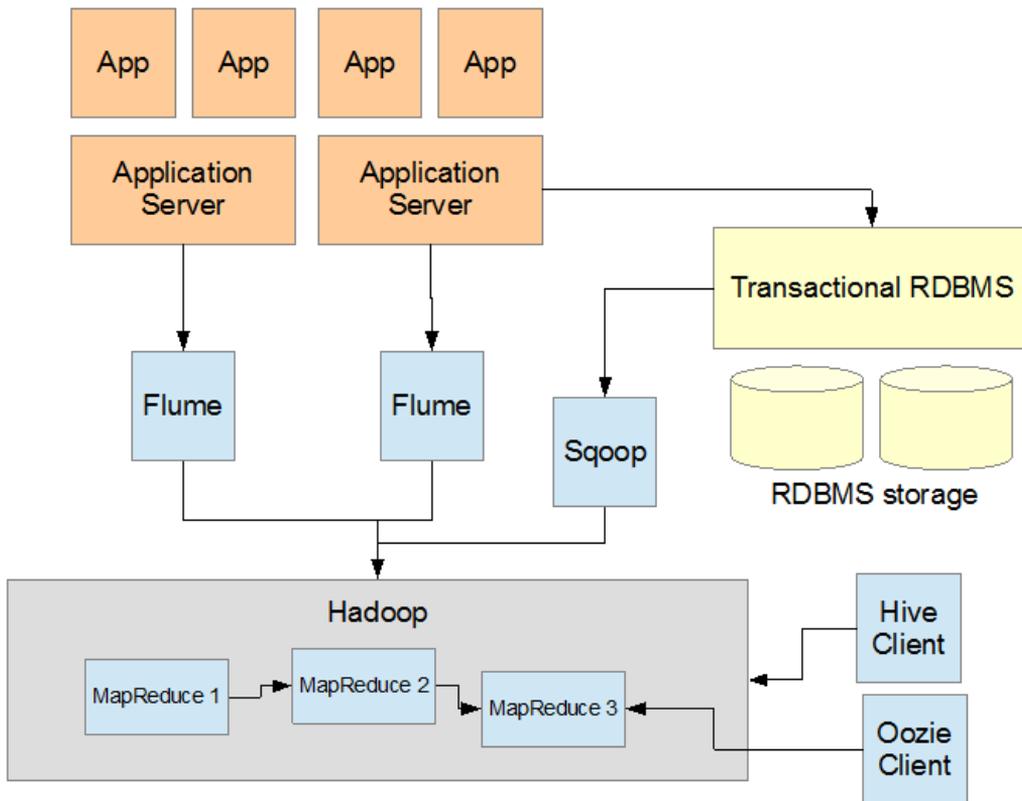


Figure 1.2 Web retail example data flow. Transactional data is retrieved from the RDBMS by Sqoop, while Flume transports log files into Hadoop.

The application servers record their log files for user interactions to their local filesystem. Transactional events may directly interact with the transactional RDBMS to process orders, although often there will be a message queue between the systems to assist with data management. Now, Hadoop's role in this architecture is to correlate the user interactions with the transactional events. For instance, if we knew the last three items the user checked out on the web site prior to making a purchase (a dress, a scarf, a handbag, then shoes, for instance) maybe we could figure out how to present combinations of these items that would help them find the right pairings. Moreover, we might be able to look at the trail of events prior to abandoning a shopping cart to reduce the rates of abandoned carts.

To do this, the transactional data has to be mined and joined together with the log data coming back from the application servers. We can use Sqoop and Flume to move these data sources onto HDFS. Once there, we almost certainly need to transform the log files into

simpler data tables, and then we can use query tools like Hive to extract the relevant sequences of events leading up to purchases and non-purchases alike.

Finally, Hadoop HDFS can serve as a repository for historical data, making it easier for us to identify longer-term patterns in the data. Because of its relatively low cost as a data analysis store, Hadoop increases the flexibility of using and storing data.

## 1.10 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.

Hadoop has really grown in the past few years with an almost overwhelming variety of supporting tools and software systems that fill-out the missing pieces of the Hadoop data story. In many cases, the best practices are evolving as the tools evolve, creating both opportunities and challenges to solving important use cases.

In the next chapter we'll go into further details about the internals of Hadoop and setting up a working Hadoop cluster.

## 1.11 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>