Hadoop IN ACTION

SECOND EDITION

Chuck P. Lam
Mark W. Davis
Ajit Gaddam

MANNING

MEAP Edition
Manning Early Access Program

Hadoop in Action

Second Edition
Version 7

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

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.[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.

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

[1] 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 2nd 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.[2]

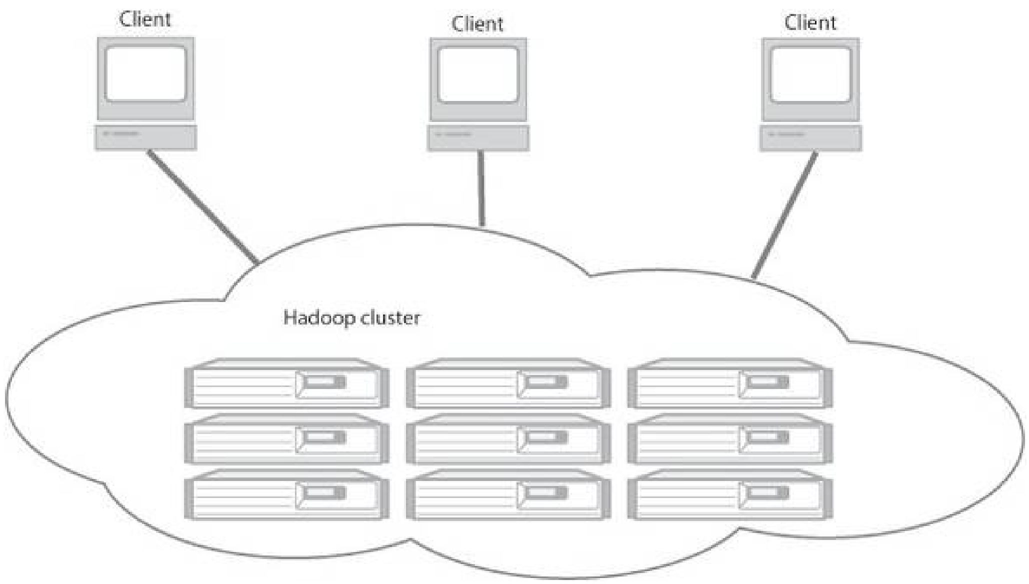


Figure 1.1 A Hadoop cluster has many parallel machines that store and process large data sets. Client computers send jobs into this computer cloud and obtain results.

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.

[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.

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.[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.

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.

[3] 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) 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.

	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 (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 `<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 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 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 keyscans 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 scheam 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

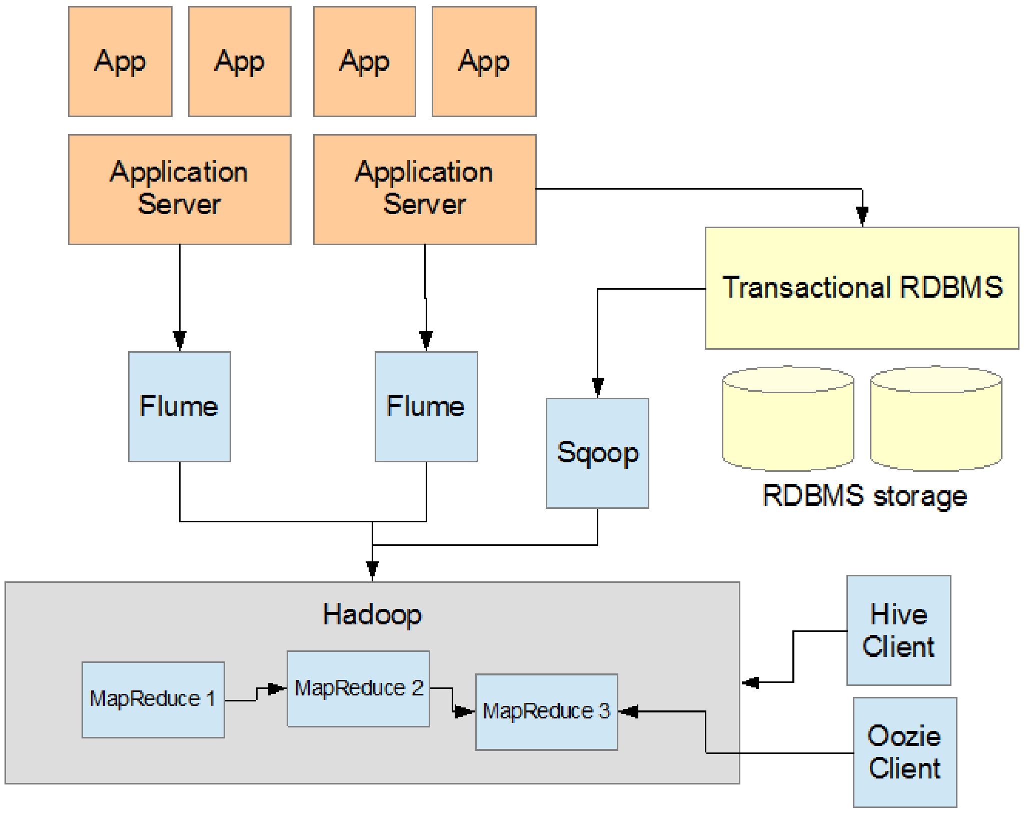


Figure 1.2 Web retail example data flow. Transactional data is retrieved from the RDBMS by Sqoop, while Flume transports log files into Hadoop.

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 their 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 applications 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 repositiory 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

2 Starting Hadoop

This chapter covers

- The architectural components of Hadoop
- Differences between Hadoop 2 and previous versions
- Setting up Hadoop and its three operating modes: standalone, pseudo-distributed, and fully distributed
- Web-based tools to monitor your Hadoop setup
- Running Hadoop in the cloud

This chapter will serve as a roadmap to guide you through setting up Hadoop. If you work in an environment where someone else sets up the Hadoop cluster for you, you may want to skim through this chapter. You'll want to understand enough to set up your personal development machine, but you can skip through the details of configuring the communication and coordination of various nodes.

After discussing the physical components of Hadoop in section 2.1, we'll progress to setting up your cluster in sections 2.2. and 2.3. Section 2.3 will focus on the three operational modes of Hadoop and how to set them up. You'll read about web-based tools that assist monitoring your cluster in section 2.4. Section 2.5 covers running Hadoop in the cloud.

2.1 The building blocks of Hadoop

We've discussed the concepts of distributed storage and distributed computation in the previous chapter. Now let's see how Hadoop implements those ideas. On a fully configured cluster, "running Hadoop" means running a set of daemons, or resident programs, on the different servers in your network. These daemons have specific roles; some exist only on one server, some exist across multiple servers. The daemons include

- NameNode

- DataNode
- Secondary NameNode
- ResourceManager
- NodeManager

We'll discuss each one and its role within Hadoop. In addition to these key components, there are two additional daemons that are run in full cluster arrangements: WebAppProxy and Job History Server. These serve supporting roles in cluster operations and are discussed further in other chapters that deal with managing a Hadoop cluster.

2.1.1 NameNode

Let's begin with arguably the most vital of the Hadoop daemons—the NameNode. Hadoop employs a master/slave architecture for both distributed storage and distributed computation. The distributed storage system is called the Hadoop File System, or HDFS. The NameNode is the master of HDFS that directs the slave DataNode daemons to perform the low-level I/O tasks. The NameNode is the bookkeeper of HDFS; it keeps track of how your files are broken down into file blocks, which nodes store those blocks, and the overall health of the distributed filesystem.

The function of the NameNode is memory and I/O intensive. As such, the server hosting the NameNode typically doesn't store any user data or perform any computations for a MapReduce or other Hadoop programs to lower the workload on the machine. This means that the NameNode server doesn't double as a DataNode.

There is unfortunately a negative aspect to the importance of the NameNode—it's a single point of failure of your Hadoop cluster. For any of the other daemons, if their host nodes fail for software or hardware reasons, the Hadoop cluster will likely continue to function smoothly or you can quickly restart it. Not so for the NameNode.

2.1.2 DataNode

Each slave machine in your cluster will host a DataNode daemon to perform the grunt work of the distributed filesystem—reading and writing HDFS blocks to actual files on the local filesystem. When you want to read or write a HDFS file, the file is broken into blocks and the NameNode will tell your client which DataNode each block resides in. Your client communicates directly with the DataNode daemons to process the local files corresponding to the blocks. Furthermore, a DataNode may communicate with other DataNodes to replicate its data blocks for redundancy.

Figure 2.1 illustrates the roles of the NameNode and DataNodes. In this figure, we show two data files, one at /user/chuck/data1 and another at /user/james/data2. The data1 file takes up three blocks, which we denote 1, 2, and 3, and the data2 file consists of blocks 4 and 5. The content of the files are distributed among the DataNodes. In this illustration, each block has three replicas. For example, block 1 (used for data1) is replicated over the three rightmost DataNodes. This ensures that if any one DataNode crashes or becomes inaccessible over the network, you'll still be able to read the files.

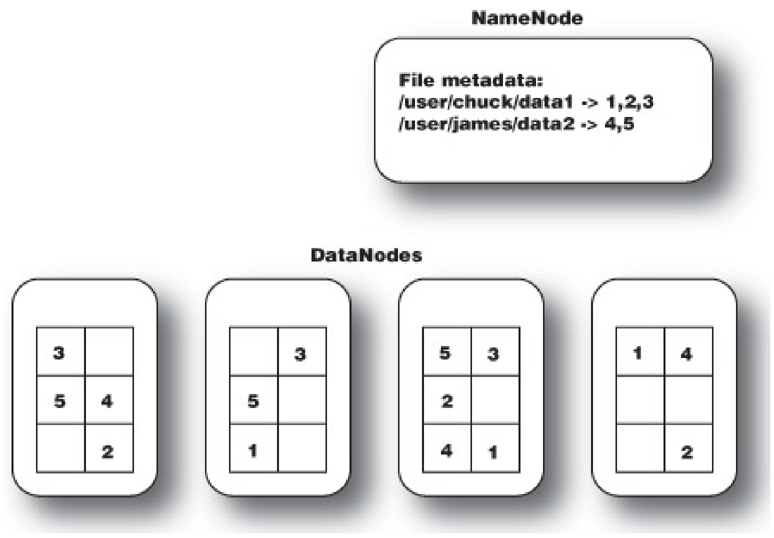


Figure 2.1 NameNode/DataNode interaction in HDFS. The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.

DataNodes are constantly reporting to the NameNode. Upon initialization, each of the DataNodes informs the NameNode of the blocks it's currently storing. After this mapping is complete, the DataNodes continually poll the NameNode to provide information regarding local changes as well as receive instructions to create, move, or delete blocks from the local disk.

2.1.3 Secondary NameNode

The Secondary NameNode (SNN) is an assistant daemon for monitoring the state of the cluster HDFS. Like the NameNode, each cluster has one SNN, and it typically resides on its own machine as well. No other DataNode of other daemons run on the same server. The SNN differs from the NameNode in that this process doesn't receive or record any real-time changes to HDFS. Instead, it communicates with the NameNode to take snapshots of the HDFS metadata at intervals defined by the cluster configuration.

As mentioned earlier, the NameNode is a single point of failure for a Hadoop cluster, and the SNN snapshots help minimize the downtime and loss of data. Nevertheless, a NameNode failure requires human intervention to reconfigure the cluster to use the SNN as the primary NameNode. We'll discuss the recovery process in chapter 8 when we cover best practices for managing your cluster.

2.1.4 ResourceManager

The ResourceManager in Hadoop 2 takes the place of JobTrackers in previous versions. The ResourceManager is part of YARN and assigns compute resources to different applications that run on the cluster. Unlike the JobTracker/TaskTracker framework, the YARN system supports applications beyond MapReduce jobs, like Spark in-memory computing and graph databases. When you create an application (or have a MapReduce job), your client contacts the ResourceManager to create an execution plan and allocate resources to the application. The ResourceManager creates ApplicationMasters that reside on DataNodes and carry out the execution of your job. It remains the liaison between your application and Hadoop.

There is only one ResourceManager daemon per Hadoop cluster. It's typically run on a server as a master node of the cluster.

2.1.5 NodeManager

A NodeManager lives on each DataNode and supplies management of the compute, I/O, and storage resources on that node. When a client requests the performance of an application like MapReduce execution, the ResourceManager requests resource allocations from the NodeManager and then contacts an ApplicationMaster. The NodeManager is responsible for creating a container for the application to run in and assigning resource "slots" to the container for use.

2.1.6 ApplicationMaster

As with the storage daemons, the computing daemons also follow a master/slave architecture: the ResourceManager is the master overseeing the overall execution of a YARN application and the ApplicationMasters manage the execution of individual tasks on each slave node. Figure 2.2 illustrates this interaction.

Each ApplicationMaster is responsible for executing the individual tasks that the ResourceManager assigns. Although there is a single ApplicationMaster per slave node, each ApplicationMaster can spawn multiple JVMs to handle many tasks in parallel.

One responsibility of the ApplicationMaster is to constantly communicate with the NodeManager and the history server. If the NodeManager fails to receive a heartbeat from a ApplicationMaster within a specified amount of time, it will assume the task has crashed and will resubmit the corresponding tasks to other nodes in the cluster.

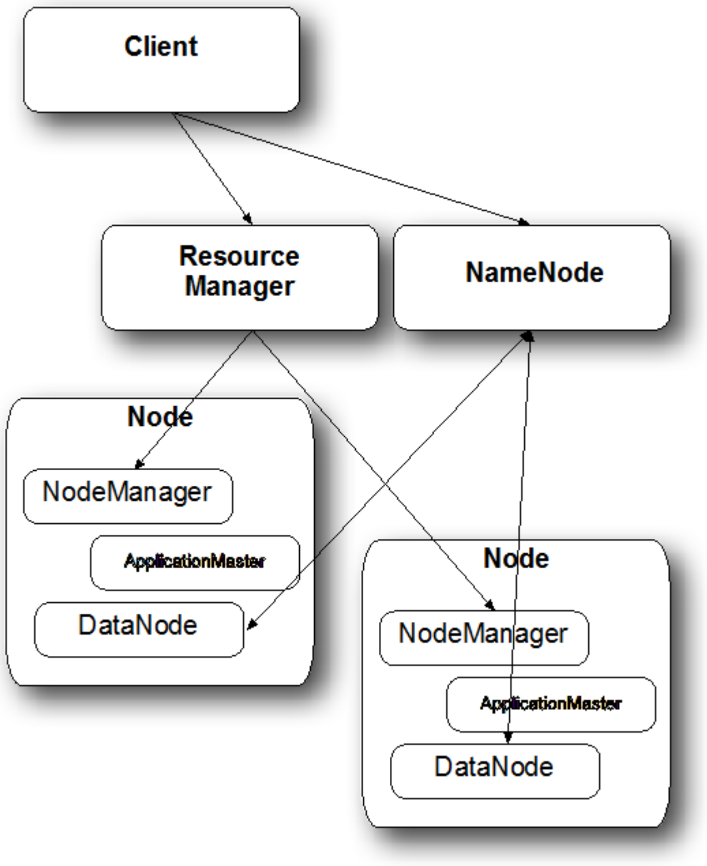


Figure 2.2 Hadoop 2 ResourceManager and node interaction. After a client calls the ResourceManager to begin a data processing job, the ResourceManager requests NodeManagers to allocate ApplicationMasters execute the tasks across the cluster.

This topology features a master node running the NameNode. Though the ResourceManager can coexist with the NameNode, best practice recommends segregating the ResourceManager to another node in the cluster. The Secondary Name Node should also be on a separate node in case the NameNode fails. For small clusters, the SNN can reside on one of the slave nodes and the ResourceManager can share the NameNode node. On the other hand, for large clusters, separate the NameNode and ResourceManager on two machines. The slave machines each host a DataNode, NodeManager, and ApplicationMaster, for running tasks on the same node where their data is stored.

We'll work toward setting up a complete Hadoop cluster of this form by first establishing the master node and the control channels between nodes. If a Hadoop cluster is already available to you, you can skip Section 2.3 on how to set up Secure Shell (SSH) channels between nodes. You also have a couple of options to run Hadoop using only a single machine, in what are known as standalone and pseudo-distributed modes. They're useful for development. Configuring Hadoop to run in these two modes or the standard cluster setup (fully distributed mode) is covered in section 2.4.

2.2 Changes in Hadoop 2

The components we've described are the parts of a Hadoop 2 cluster. These differ from previous versions of Hadoop and represent a significant change in the way Hadoop works. Most notably, in previous versions of Hadoop a JobTracker managed MapReduce jobs that were sent to TaskTrackers on DataNodes. With Hadoop 2, however, the JobTracker and TaskTracker are replaced by the ResouceManager and ApplicationMaster components. The NodeManager serves an additional role by supporting resource allocation for application containers. MapReduce is now a YARN application in Hadoop 2 but otherwise behaves exactly the way it previously did.

2.3 Setting up SSH for a Hadoop cluster

When setting up a Hadoop cluster, you'll need to designate one specific node as the master node. As shown in figure 2.3, this server will typically host the NameNode and JobTracker daemons. It'll also serve as the base station contacting and activating the DataNode and TaskTracker daemons on all of the slave nodes. As such, we need to define a means for the master node to remotely access every node in your cluster.

Hadoop uses passphraseless SSH for this purpose. SSH utilizes standard public key cryptography to create a pair of keys for user verification—one public, one private. The public key is stored locally on every node in the cluster, and the master node sends the private key when attempting to access a remote machine. With both pieces of information, the target machine can validate the login attempt.

2.3.1 Define a common account

We've been speaking in general terms of one node accessing another; more precisely this access is from a user account on one node to another user account on the target machine. For Hadoop, the accounts should have the same username on all of the nodes (we use hadoop-user in this book), and for security purpose we recommend it being a user-level account. This account is only for managing your Hadoop cluster. Once the cluster daemons are up and running, you'll be able to run your actual MapReduce jobs from other accounts.

2.3.2 Verify SSH installation

The first step is to check whether SSH is installed on your nodes. We can easily do this by use of the `which` UNIX command:

```
[hadoop-user@master]$ which ssh
/usr/bin/ssh

[hadoop-user@master]$ which sshd
```

```
/usr/bin/sshd

[hadoop-user@master]$ which ssh-keygen
/usr/bin/ssh-keygen
```

If you instead receive an error message such as this,

```
/usr/bin/which: no ssh in (/usr/bin:/bin:/usr/sbin...
```

install OpenSSH (www.openssh.com) via a Linux package manager or by downloading the source directly. (Better yet, have your system administrator do it for you.)

2.3.3 Generate SSH key pair

Having verified that SSH is correctly installed on all nodes of the cluster, we use `sshkeygen` on the master node to generate an RSA key pair. Be certain to avoid entering a passphrase, or you'll have to manually enter that phrase every time the master node attempts to access another node:

```
[hadoop-user@master]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hadoop-user/.ssh/id_rsa): Enter passphrase
      (empty for no passphrase):
Enter same passphrase again:

Your identification has been saved in /home/hadoop-user/.ssh/id_rsa. Your public key
      has been saved in /home/hadoop-user/.ssh/id_rsa.pub.
```

After creating your key pair, your public key will be of the form

```
[hadoop-user@master]$ more /home/hadoop-user/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA1WS3RG8LrZH4zL2/1oYgkV1OmVclQ2OO5vRi0Nd
      K51Sy3wWpBVHx82F3x3ddoZQjBK3uvLMaDhXvncJG31JPfU7CTAfmtgINYv0kdUbDJq4TKG/fuO5q
      J9CqHV71thN2M310gcJ0Y9YCN6grmsiWb2iMcXpy2pqg8UM3ZKApyIPx99O1vREWm+4moFTg
      YwIl5be23ZCyxNjgZFWk5MRlT1p1TxB68jqNbPQtU7fIafS7Sasy7h4eyIy7cbLh8x0/V4/mcQsY
5dvReitNvFVte6onl8YdmnMpAh6nwCvog3UeWWJjVZTEBFkTZuV1i9HeYHxpm1wAzcnf7az78jT IRQ==
      hadoop-user@master
```

We next need to distribute this public key across your cluster.

2.3.4 Distribute public key and validate logins

Albeit a bit tedious, you'll next need to copy the public key to every slave node as well as the master node:

```
[hadoop-user@master]$ scp ~/.ssh/id_rsa.pub hadoop-user@target:~/master_key
```

Manually log in to the target node and set the master key as an authorized key (or append to the list of authorized keys if you have others defined).

```
[hadoop-user@target]$ mkdir ~/.ssh
[hadoop-user@target]$ chmod 700 ~/.ssh
[hadoop-user@target]$ mv ~/master_key ~/.ssh/authorized_keys
[hadoop-user@target]$ chmod 600 ~/.ssh/authorized_keys
```

After generating the key, you can verify it's correctly defined by attempting to log in to the target node from the master:

```
[hadoop-user@master]$ ssh target
The authenticity of host 'target (xxx.xxx.xxx.xxx)' can't be established. RSA key
      fingerprint is 72:31:d8:1b:11:36:43:52:56:11:77:a4:ec:82:03:1d. Are you sure
      you want to continue connecting (yes/no)? yes
Warning: Permanently added 'target' (RSA) to the list of known hosts. Last login: Sun
      Jan 4 15:32:22 2009 from master
```

After confirming the authenticity of a target node to the master node, you won't be prompted upon subsequent login attempts.

```
[hadoop-user@master]$ ssh target
Last login: Sun Jan 4 15:32:49 2009 from master
```

We've now set the groundwork for running Hadoop on your own cluster. Let's discuss the different Hadoop modes you might want to use for your projects.

2.4 Running Hadoop

We need to configure a few things before running Hadoop. Let's take a closer look at the Hadoop configuration directory:

```
[hadoop-user@master]$ cd $HADOOP_HOME
[hadoop@hadoop hadoop-2.4.1]$ ls -l etc/hadoop/
total 124
-rw-r--r--. 1 hadoop hadoop  3589 Jun 20 23:05 capacity-scheduler.xml
-rw-r--r--. 1 hadoop hadoop  1335 Jun 20 23:05 configuration.xsl
-rw-r--r--. 1 hadoop hadoop   318 Jun 20 23:05 container-executor.cfg
-rw-r--r--. 1 hadoop hadoop   774 Jun 20 23:05 core-site.xml
-rw-r--r--. 1 hadoop hadoop  3589 Jun 20 23:05 hadoop-env.cmd
-rw-r--r--. 1 hadoop hadoop  3494 Jun 20 23:05 hadoop-env.sh
-rw-r--r--. 1 hadoop hadoop  1774 Jun 20 23:05 hadoop-metrics2.properties
-rw-r--r--. 1 hadoop hadoop  2490 Jun 20 23:05 hadoop-metrics.properties
-rw-r--r--. 1 hadoop hadoop  9257 Jun 20 23:05 hadoop-policy.xml
-rw-r--r--. 1 hadoop hadoop   775 Jun 20 23:05 hdfs-site.xml
-rw-r--r--. 1 hadoop hadoop  1449 Jun 20 23:05 httpfs-env.sh
-rw-r--r--. 1 hadoop hadoop  1657 Jun 20 23:05 httpfs-log4j.properties
-rw-r--r--. 1 hadoop hadoop    21 Jun 20 23:05 httpfs-signature.secret
-rw-r--r--. 1 hadoop hadoop   620 Jun 20 23:05 httpfs-site.xml
-rw-r--r--. 1 hadoop hadoop 11169 Jun 20 23:05 log4j.properties
-rw-r--r--. 1 hadoop hadoop   918 Jun 20 23:05 mapred-env.cmd
-rw-r--r--. 1 hadoop hadoop  1383 Jun 20 23:05 mapred-env.sh
-rw-r--r--. 1 hadoop hadoop  4113 Jun 20 23:05 mapred-queues.xml.template
-rw-r--r--. 1 hadoop hadoop   758 Jun 20 23:05 mapred-site.xml.template
-rw-r--r--. 1 hadoop hadoop    10 Jun 20 23:05 slaves
-rw-r--r--. 1 hadoop hadoop  2316 Jun 20 23:05 ssl-client.xml.example
-rw-r--r--. 1 hadoop hadoop  2268 Jun 20 23:05 ssl-server.xml.example
-rw-r--r--. 1 hadoop hadoop  2178 Jun 20 23:05 yarn-env.cmd
-rw-r--r--. 1 hadoop hadoop  4564 Jun 20 23:05 yarn-env.sh
-rw-r--r--. 1 hadoop hadoop   690 Jun 20 23:05 yarn-site.xml
```

The first thing you need to do is to specify the location of Java on all the nodes including the master. In hadoop-env.sh and yarn-env.sh define the `JAVA_HOME` environment variable to point to the Java installation directory. On our servers, we've it defined as

```
export JAVA_HOME=/usr/share/jdk
```

(If you followed the examples in chapter 1, you've already completed this step.)

The hadoop-env.sh and yarn-env.sh files contain other variables for defining your Hadoop environment, but `JAVA_HOME` is the only one requiring initial modification. The default settings on the other variables will probably work fine. As you become more familiar with Hadoop you can later modify this file to suit your individual needs (logging directory location, Java class path, and so on).

The majority of Hadoop settings are contained in XML configuration files. Before version 0.20, these XML files were hadoop-default.xml and hadoop-site.xml. As the names imply, hadoop-default.xml contained the default Hadoop settings to be used unless they were explicitly overridden in hadoop-site.xml. In practice you only dealt with hadoop-site.xml. In version 0.20 this file was separated into three XML files: core-site.xml, hdfs-site.xml, and mapred-site.xml. This refactoring better aligned the configuration settings to the subsystem of Hadoop that they control. With Hadoop 2, additional new configuration files were added, most notably yarn-site.xml that configures the YARN system. In the rest of this chapter we'll generally point out which of the files are used to adjust a configuration setting. If you use an earlier version of Hadoop, keep in mind that all such configuration settings are modified in other locations.

In the following subsections we'll provide further details about the different operational modes of Hadoop and example configuration files for each.

2.4.1 Local (standalone) mode

The standalone mode is the default mode for Hadoop. When you first uncompress the Hadoop source package, it's ignorant of your hardware setup. Hadoop chooses to be conservative and assumes a minimal configuration. All of the XML files are empty under this default mode. For instance, core-site.xml looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<!--   Licensed under the Apache License… -->
<!-- Put site-specific property overrides in this file. -->
<configuration> </configuration>
```

With empty configuration files, Hadoop will run completely on the local machine. Because there's no need to communicate with other nodes, the standalone mode doesn't use HDFS, nor will it launch any of the Hadoop daemons. In fact, any program output will be written to the local filesystem. Standalone mode's primary use is for developing and debugging the application logic of a MapReduce program without the additional complexity of interacting with the daemons. When you ran the example MapReduce program in chapter 1, you were running it in standalone mode.

2.4.2 Pseudo-distributed mode

The pseudo-distributed mode is running Hadoop in a "cluster of one" with a combination of daemons running on a single machine. This mode complements the standalone mode for debugging your application, allowing you to examine memory usage, HDFS input/output issues, and other daemon interactions. Listing 2.1 provides simple XML files to configure a single server in this mode. Prior to configuring Hadoop and starting it, HADOOP_HOME and some related variables should be set at the command line (or in the hadoop-user .bashrc file):

```
export HADOOP_HOME=path_to_hadoop
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
```

The HADOOP_COMMON_LIB_NATIVE_DIR variable provides the path for native platform libraries for all Hadoop scripts, and is important for the operation of those scripts. The required configuration changes listed below do not include the configuration of YARN, which is covered in the following section, so any MapReduce programs will be run in "local" mode but the results will be placed in HDFS rather than the local filesystem.

Listing 2.1 Example of configuration files for pseudo-distributed mode

```
core-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
 <name>fs.defaultFS</name>
 <value>hdfs://localhost:9000</value>
 <description>The name of the default file system. A URI whose
 scheme and authority determine the FileSystem implementation.
 </description>
</property>

</configuration>

hdfs-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
 <name>dfs.replication</name>
 <value>1</value>
 <description>The actual number of replications can be specified when the
 file is created.</description>
</property>

</configuration>
```

In core-site.xml we specify the hostname and port of the NameNode (note that the Hadoop 2 property is fs.defaultFS rather than fs.default.name in previous versions). In hdfs-site.xml we specify the default replication factor for HDFS, which should only be one because we're running on only one node.

While all the daemons are running on the same machine, they still communicate with each other using the same SSH protocol as if they were distributed over a cluster. Section 2.2 has a more detailed discussion of setting up the SSH channels, but for single-node operation simply check to see if your machine already allows you to ssh back to itself.

```
[hadoop-user@master]$ ssh localhost
```

If it does, then you're good. Otherwise setting up takes two lines.

```
[hadoop-user@master]$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
[hadoop-user@master]$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

You are almost ready to start Hadoop. But first you'll need to format your HDFS by using the command

```
[hadoop-user@master]$ bin/hadoop namenode -format
```

We can now launch the distributed file system daemons by use of the sbin/start-dfs.sh script. The Java `jps` command will list all daemons to verify the setup was successful.

```
[hadoop-user@master]$ bin/start-all.sh
[hadoop-user@master]$ jps
10162 SecondaryNameNode
10007 DataNode
9883 NameNode
10631 Jp
```

You can also test the operation of your Hadoop system by running a command:

```
[hadoop-user@master]$ bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-
      examples-2.X.X.jar randomwriter out2
```

In this example, we are executing the randomwriter sample MapReduce program that will generate 10g of random data suitable for testing MapReduce sort.

When you've finished with pseudo-distributed Hadoop you can shut down the Hadoop daemons by the command

```
[hadoop-user@master]$ sbin/stop-dfs.sh
```

Finally, pseudo-distributed mode can also be configured to use YARN as the MapReduce implementation framework. Listing 2.2 shows the additional configuration changes needed for YARN:

Listing 2.2 Example YARN configuration for pseudo-distributed mode

```
yarn-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
     <property>
     <name>yarn.nodemanager.aux-services</name>
     <value>mapreduce_shuffle</value>
     </property>
</configuration>
```

After configuring YARN, start the YARN daemons using sbin/start-yarn.sh following starting the DFS daemons.

Both standalone and pseudo-distributed modes are for development and debugging purposes. An actual Hadoop cluster runs in the third mode, the fully distributed mode.

2.4.3 Fully distributed mode

After continually emphasizing the benefits of distributed storage and distributed computation, it's time for us to set up a full cluster. In the discussion below we'll use the following server names:

- *master*—The master node of the cluster and host of the NameNode and ResourceManager daemons
- *backup*—The server that hosts the Secondary NameNode daemon
- *hadoop1, hadoop2, hadoop3, ...*—The slave boxes of the cluster running both DataNode and NodeManager daemons. ApplicationMasters will also run on these machines.

Using the preceding naming convention, listing 2.3 is a modified version of the pseudo-distributed configuration files (listings 2.1 and 2.2) that can be used as a skeleton for your cluster's setup.

Listing 2.3 Example configuration files for fully distributed mode

```
core-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
 <name>fs.defaultFS</name>                                  1
 <value>hdfs://master:9000</value>
 <description>The name of the default file system. A URI whose scheme
 and authority determine the FileSystem implementation.
 </description>
</property>

</configuration>

yarn-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>
     <property>
     <name>yarn.nodemanager.aux-services</name>
     <value>mapreduce_shuffle</value>
     </property>
</configuration>

hdfs-site.xml
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>
 <name>dfs.replication</name>                                    3
 <value>3</value>
 <description>The actual number of replications can be specified when the
 file is created.</description>
</property>

</configuration>
```

The key differences between these new modifications and the pseudo-distributed equivalents are

- We explicitly stated the hostname for location of the NameNode #1 daemon.
- We increased the HDFS replication factor to take advantage of distributed storage. Recall that data is replicated across HDFS to increase availability and reliability.

We also need to update the masters and slaves files to reflect the locations of the other daemons. Edit etc/hadoop/slaves to include the three new cluster machines, hadoop1, hadoop2 and hadoop3.

Once you have copied these files across all the nodes in your cluster, be sure to format HDFS to prepare it for storage:

```
[hadoop-user@master]$ bin/hadoop namenode -format
```

Now you can start the Hadoop daemons:

```
[hadoop-user@master]$ sbin/start-all.sh
```

and verify the nodes are running their assigned jobs.

```
[hadoop-user@master]$ jps
12416 Jps
9883 NameNode
11302 ResourceManager

[hadoop-user@backup]$ jps
2099 Jps
1679 SecondaryNameNode

[hadoop-user@hadoop1]$ jps
11398 NodeManager
12416 Jps
10007 DataNode
```

You have a functioning cluster!

Switching between modes

A practice that we found useful when starting with Hadoop was to use symbolic links to switch between Hadoop modes instead of constantly editing the XML files. To do so, create a separate configuration folder for each of the modes and place the appropriate version of the XML files in the corresponding folder. Below is an example directory listing:

```
[hadoop@hadoop_master hadoop]$ ls -l total 4884
drwxr-xr-x 2 hadoop-user hadoop 4096 Nov 26 17:36 bin

-rw-rw-r- -  1 hadoop-user hadoop 57430 Nov 13 19:09 build.xml drwxr-xr-x 4 hadoop-
user hadoop 4096 Nov 13 19:14 c++
-rw-rw-r- -  1 hadoop-user hadoop 287046 Nov 13 19:09 CHANGES.txt lrwxrwxrwx 1
hadoop-user hadoop  12 Jan 5 16:06 conf -> conf.cluster drwxr-xr-x 2 hadoop-user
hadoop 4096 Jan 8 17:05 conf.cluster
drwxr-xr-x 2 hadoop-user hadoop 4096 Jan 2 15:07 conf.pseudo drwxr-xr-x 2 hadoop-user
hadoop 4096 Dec 1 10:10 conf.standalone drwxr-xr-x 12 hadoop-user hadoop 4096 Nov 13
19:09 contrib drwxrwxr-x 5 hadoop-user hadoop 4096 Jan 2 09:28 datastore
drwxr-xr-x 6 hadoop-user hadoop 4096 Nov 26 17:36 docs
...
```

You can then switch between configurations by using the Linux `ln` command (e.g., `ln -s conf.cluster conf`). This practice is also useful to temporarily pull a node out of the cluster to debug a MapReduce program in pseudo-distributed mode, but be sure that the modes have different file locations for HDFS and stop all daemons on the node before changing configurations.

Now that we've gone through all the settings to successfully get a Hadoop cluster up and running, we'll introduce the Web UI for basic monitoring of the cluster's state.

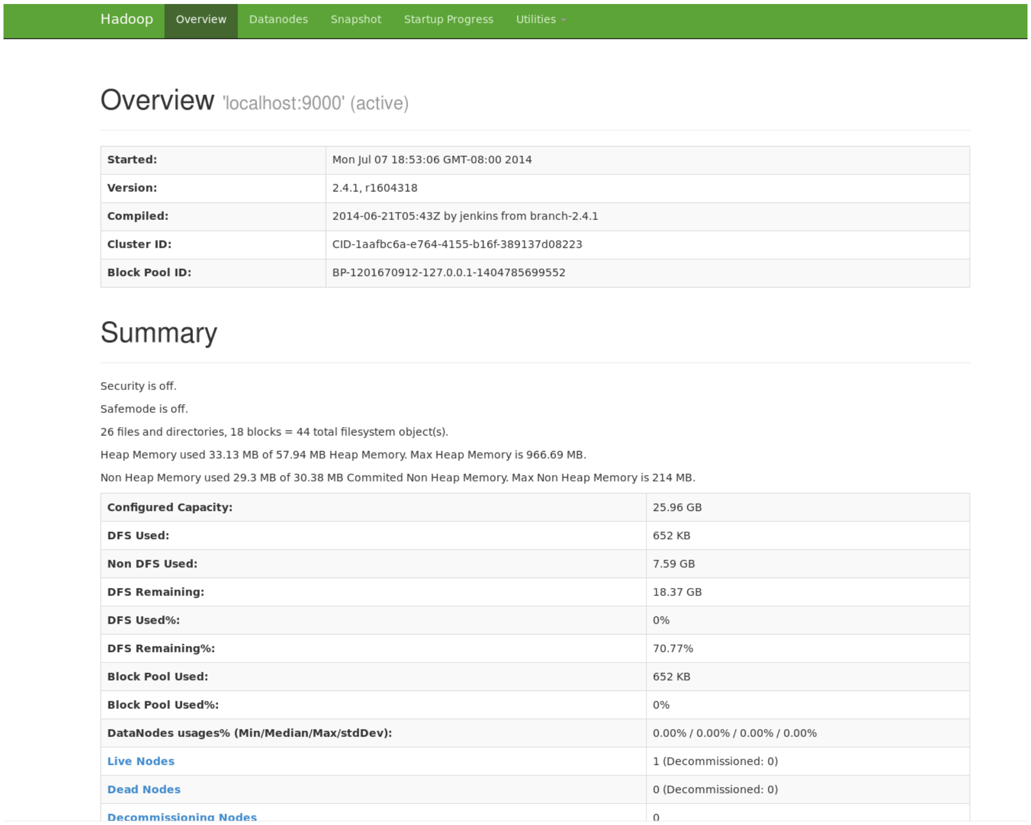
2.5 Web-based cluster UI

Having covered the operational modes of Hadoop, we can now introduce the web interfaces that Hadoop provides to monitor the health of your cluster. The browser interface allows you to access information you desire much faster than digging through logs and directories.

The NameNode hosts a general report on port 50070. It gives you an overview of the state of your cluster's HDFS. Figure 2.4 displays this report for a 1-node cluster example. From this interface, you can browse through the filesystem, check the status of each DataNode in your cluster, and peruse the Hadoop daemon logs to verify your cluster is functioning correctly.

Hadoop provides a similar status overview of cluster health and applications. Figure 2.5 depicts one hosted at port 8088 of the ResourceManager node.

Again, a wealth of information is available through this reporting interface. You can access the status of ongoing Applications as well as detailed reports about completed jobs. The latter is of particular importance—these logs describe which nodes performed which tasks and the time/resources required to complete each task.



Hadoop Overview Datanodes Snapshot Startup Progress Utilities

Overview 'localhost:9000' (active)

Started:	Mon Jul 07 18:53:06 GMT-08:00 2014
Version:	2.4.1, r1604318
Compiled:	2014-06-21T05:43Z by jenkins from branch-2.4.1
Cluster ID:	CID-1aafbc6a-e764-4155-b16f-389137d08223
Block Pool ID:	BP-1201670912-127.0.0.1-1404785699552

Summary

Security is off.

Safemode is off.

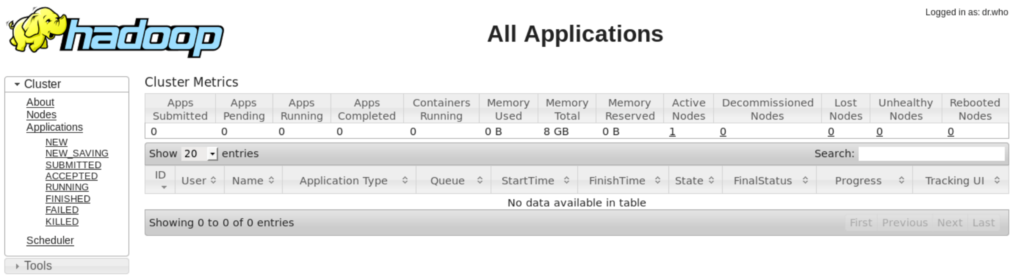
26 files and directories, 18 blocks = 44 total filesystem object(s).

Heap Memory used 33.13 MB of 57.94 MB Heap Memory. Max Heap Memory is 966.69 MB.

Non Heap Memory used 29.3 MB of 30.38 MB Commited Non Heap Memory. Max Non Heap Memory is 214 MB.

Configured Capacity:	25.96 GB
DFS Used:	652 KB
Non DFS Used:	7.59 GB
DFS Remaining:	18.37 GB
DFS Used%:	0%
DFS Remaining%:	70.77%
Block Pool Used:	652 KB
Block Pool Used%:	0%
DataNodes usages% (Min/Median/Max/stdDev):	0.00% / 0.00% / 0.00% / 0.00%
Live Nodes	1 (Decommissioned: 0)
Dead Nodes	0 (Decommissioned: 0)
Decommissioning Nodes	0

Figure 2.4 A snapshot of the HDFS web interface. From this interface you can browse through the HDFS filesystem, determine the storage available on each individual node, and monitor the overall health of your cluster.



Logged in as: dr.who

All Applications

Cluster
About
Nodes
Applications
NEW
NEW_SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED
Scheduler
Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
0	0	0	0	0	0 B	8 GB	0 B	1	0	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
No data available in table										

Showing 0 to 0 of 0 entries First Previous Next Last

Figure 2.5 A snapshot of the cluster management web interface. This tool allows you to monitor active applications and access the logs of each task. The logs of previously submitted jobs are also available (under Tools) and are useful for debugging your programs.

Though the usefulness of these tools may not be immediately apparent at this stage, they'll come in handy as you begin to perform more sophisticated tasks on your cluster. You'll realize their importance as we study Hadoop more in depth.

2.6 Running Hadoop in the cloud

Cloud-hosted Hadoop is appealing to many application developers and data scientists who are not able to build local clusters. It also makes sense for data processing when the data already resides in cloud storage systems. In this section we will provide an overview of how to run Hadoop in Amazon Web Services (AWS) infrastructure. While much of the information is tailored to the AWS, some of the material can also assist when trying to run Hadoop in other cloud infrastructure.

2.6.1 Introducing Amazon Web Services

Amazon Web Services are a suite of capabilities for storing information in the cloud, retrieving the information, creating compute instances, and performing other tasks. AWS began with just a few capabilities but has expanded over the years to move beyond Infrastructure as a Service (IaaS) to also include Platform as a Service (PaaS) capabilities like data storage and retrieval, caching, and other features.

One key feature of AWS is Amazon Elastic Map Reduce or EMR. EMR is Hadoop that runs on the AWS infrastructure and that can easily add or subtract nodes that dynamically start in order to efficiently run MapReduce jobs.

2.6.2 Setting up AWS

To get started with AWS, you need to create an account at aws.amazon.com. You will need to supply credit card information to pay for your usage of AWS services. Note that AWS is not free and you should be careful to monitor your use of the services to avoid being blindsided by unexpected charges.

Once your account is ready, you will need to create at least one Elastic Compute Cloud (EC2) application key pair. Click on the EC2 service panel and then on Key Pair to create a new key pair. Name it something memorable. The new PEM file will download to your local machine. Save that file for future use. If you lose it, you will not be able to recover it but will have to make a new key pair.

2.6.3 Setting up EMR

Setting up EMR can be a bit confusing, but there are a few basics that conceptually simplify the way the system works. First, EMR gets its input data from Amazon Simple Storage Service (S3), writes its results back to S3, and even logs progress into S3. S3 is as simple as its name suggests; you create a "bucket" and the bucket has folders. You pay a few cents per month for storing small data results in S3. Before you begin with EMR, you should create a bucket or two to store EMR results and logs. Click the S3 icon and then select Create Bucket. Give the bucket names like "my-emr-logs" and "my-emr-results" to get started.

Second, there are already sample Hadoop workflows available in EMR, including performing a wordcount over sample data, contextual advertising, and processing log files using Pig scripting. The easiest way to get started is to just use one of the samples. To do so, select "Create Cluster" from the EMR control panel and then select "Configure Sample Application" from the configuration screen. The only other information that you need to supply are the names of the S3 buckets that we created above and the EC2 key pair. You can optionally also set the version of Hadoop you want to use and some other tags for later discovery.

When you create the cluster, it will begin provisioning the compute nodes to run your sample job. This can take a few minutes while AWS builds the Hadoop instances in their respective roles and imports the necessary job-related files. Running the job will typically take a few minutes as well.

2.6.4 Running MapReduce jobs on EMR

In addition to the sample workflows that EMR has available for clusters, you can also run customer MapReduce code on EMR. When we get to MapReduce programming you will learn more about how to write, compile, and package your own MapReduce programs for execution on Hadoop. For EMR, the only remaining step is to upload the Java jars into S3 and then configure a "step" in the EMR process to use your custom jar. You will also typically transfer your input data to an S3 bucket and that input path will be specified for the EMR program to use as input to the MapReduce program.

2.6.5 Shutting down AWS and EMR

The cool thing about EMR is that it is "elastic" and will terminate the compute cluster after you finish your task. There is really nothing more to do. Note, though, that you will be charged for the S3 storage that you use. It's pretty cheap, though, compared with running compute nodes. Delete S3 buckets to reduce your costs.

2.7 Summary

In this chapter we've discussed the key nodes and the roles they play within the Hadoop architecture. You've learned how to configure your cluster, as well as manage some basic tools to monitor your cluster's overall health. Finally, you learned the basics of using Hadoop in the Amazon cloud via Elastic Map Reduce or EMR, that lets you drive Hadoop without needing to manage your own clusters.

Overall, this chapter focuses on one-time tasks. Once you've formatted the NameNode for your cluster, you'll (hopefully) never need to do so again. Likewise, you shouldn't keep altering the core-site.xml configuration file for your cluster or assigning daemons to nodes. In the next chapter, you'll learn about the aspects of Hadoop you'll be interacting with on a daily basis, such as managing files in HDFS. With this knowledge you'll be able to begin writing your own MapReduce applications and realize the true potential that Hadoop has to offer.

3 Securing the Hadoop Platform

Security and privacy issues are magnified by the volume, variety, and velocity of Big Data. The diversity of data sources, formats, and data flows, combined with the streaming nature of data acquisition and high volume create unique security risks. Security and privacy issues are magnified by the volume, variety, and velocity of Big Data. The diversity of data sources, formats, and data flows, combined with the streaming nature of data acquisition and high volume create unique security risks.

It is not merely the existence of large amounts of data that is creating new security challenges for organizations. Big Data has been collected and utilized by enterprises for several decades. Software infrastructures such as Hadoop enable developers and analysts to easily leverage hundreds of computing nodes to perform data-parallel computing which was not there before. As a result, new security challenges have arisen from the coupling of Big Data with heterogeneous compositions of commodity hardware with commodity operating systems, and commodity software infrastructures for storing and computing on data. As Big Data expands at the different enterprises, traditional security mechanisms tailored to securing small-scale, static data and data flows on firewalled and semi-isolated networks are inadequate. Similarly, it is unclear how to retrofit provenance in an enterprise's existing infrastructure.

In this chapter, we will detail the security challenges involved when organizations start moving sensitive data to a Big Data repository like Hadoop. We will also identify the different threat models and build the security control framework to address and mitigate security risks due to the identified threat conditions and usage models.

3.1 Hadoop Security Weaknesses

Traditional Relational Database Management Systems (RDBMS) security has evolved over the years and with many 'eyeballs' assessing the security through various security evaluations.

Unlike such solutions, Hadoop security has not undergone the same level of rigor or evaluation for that matter and thus can claim little assurance of the implemented security.

Another big challenge is that today, there is no standardization or portability of security controls between the different Open-Source Software (OSS) projects and the different Hadoop or Big Data vendors. Hadoop security is completely fragmented. This is true even when the above parties implement the same security feature for the same Hadoop component and usually results in force-fitting security into the Apache Hadoop framework.

3.1.1 Top 10 Security and Privacy Challenges in Hadoop

The Cloud Security Alliance Big Data Security Working Group has compiled the following as the Top 10 security and privacy challenges to overcome in Big Data:

1. Secure computations in distributed programming frameworks
2. Security best practices for non-relational data stores
3. Secure data storage and transactions logs
4. End-point input validation/filtering
5. Real-time security monitoring
6. Scalable privacy-preserving data mining and analytics
7. Cryptographically enforced data centric security
8. Granular access control
9. Granular audits
10. Data provenance

The above challenges were grouped into four broad components by the Cloud Security Alliance. They were:

Infrastructure Security

- Secure computations in distributed programming frameworks
- Security best practices for non-relational data stores

Data Privacy

- Scalable privacy-preserving data mining and analytics
- Cryptographically enforced data centric security
- Granular access control

Data Management

- Secure data storage and transactions logs
- Granular audits
- Data provenance

Integrity & Reactive Security

- End-point input validation/filtering

- Real-time security monitoring

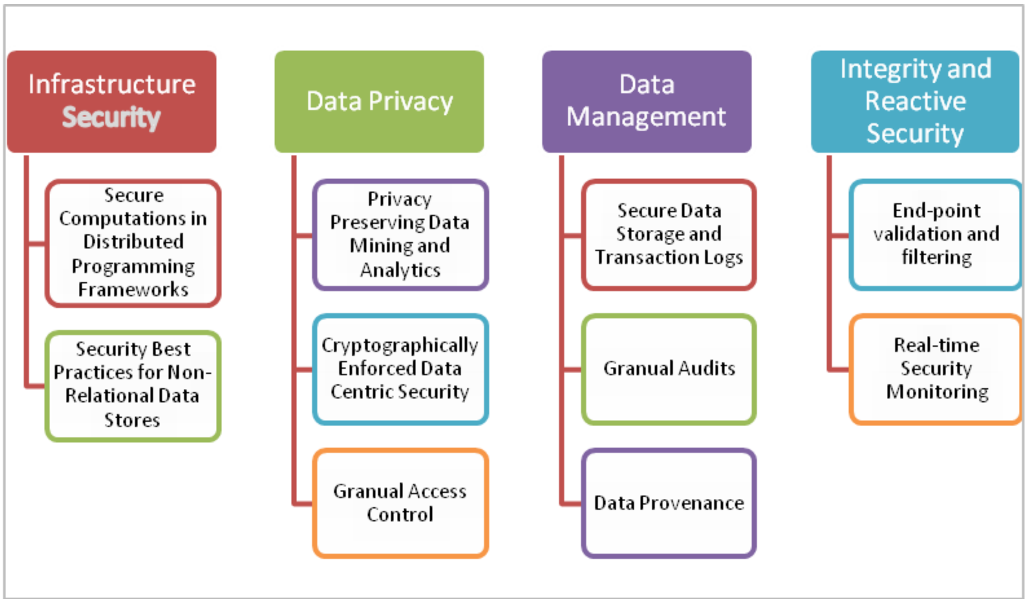

Infrastructure Security
Secure Computations in Distributed Programming Frameworks
Security Best Practices for Non-Relational Data Stores
Data Privacy
Privacy Preserving Data Mining and Analytics
Cryptographically Enforced Data Centric Security
Granual Access Control
Data Management
Secure Data Storage and Transaction Logs
Granual Audits
Data Provenance
Integrity and Reactive Security
End-point validation and filtering
Real-time Security Monitoring

Figure 3.1 CSA Classification of the Top 10 Security Challenges in Hadoop

3.1.2 Additional Security Weaknesses

The earlier section regarding Cloud Security Alliance list is an excellent start and this research and paper significantly adds to it. Where possible, effort has been made to map back to the categories identified in the CSA work. This section lists some additional security weaknesses associated with Open Source Software (OSS) like Apache Hadoop. It is meant to give the reader an idea of the possible attack surface. However, it is not meant to be exhaustive which subsequent sections will provide and add to.

Infrastructure Security & Integrity

- The Common Vulnerabilities and Exposures (CVE) database only shows four reporting and fixed Hadoop vulnerabilities over the past three years. Software, even Hadoop, is far from perfect. This could reflect either that the security community is not active or that most of vulnerability remediation happens internally within the vendor environments themselves with no public reporting.
- Hadoop security configuration files are not self-contained with no validity checks prior to such policies being deployed. This usually results in data integrity and availability issues.

Identity & Access Management

- Role Based Access Control (RBAC) policy files and Access Control Lists (ACLs) for components like MapReduce and HBase are usually configured via clear-text files. These files are editable by privileged accounts on the system like *root* and other application accounts.

Data Privacy & Security

- Not all issues associated with SQL injection type of attacks go away. They move with Hadoop components like Hive and Impala. SQL prepare functions are currently not available which would have enabled separation of the query and data
- Lack of native cryptographic controls for sensitive data protection. Frequently, such security is provided outside the data or application stack.
- Clear-text data might be sent when communicating between DataNode to DataNode since data locality cannot be strictly enforced and the scheduler might not be able to find resources next to the data and force it to read data over the network.

3.2 Hadoop Threat Model

3.2.1 Challenges and Threats in Hadoop Security

Table 1: Challenges and Threats in Hadoop Security

Challenge Category	Challenge	Description	Threat Scenarios	Solution/security control attributes
Infrastructure Security	Secure computations in distributed programming frameworks	Distributed programming frameworks utilize parallel computation and storage to process massive amounts of data. Untrusted mappers can be altered to snoop on requests, alter MapReduce scripts, or alter results. The most difficult	• Malfunctioning data nodes • Infrastructure attacks • Rogue data nodes	• Trustworthiness between nodes • Access Control To establish trust and access control between nodes, almost often, the reliance is either on network based controls to control the relationship between the management nodes and the data nodes to create a 1-1 mapping. Using nodes based on SE Linux further the capability to impose a Mandatory Access Control [MAC] policy that can mitigate this threat

Challenge Category	Challenge	Description	Threat Scenarios	Solution/security control attributes
		problem is to detect mappers returning incorrect results, which will, in turn, generate incorrect aggregate outputs. With large data sets, it is nearly impossible to identify malicious mappers that may create significant damage, especially for scientific and financial computations.		
	Security Best Practices for Non-Relational Data Stores	The security infrastructures of non-relational data stores popularized by NoSQL databases are still evolving. NoSQL databases accommodate and process huge volumes of static and streaming data	• Lax and insufficient authentication & authorization mechanisms • Susceptibility to injection attacks such as JSON and REST injection	• Enforcing data integrity either in the data store or at the application/middleware layer • Securing passwords & credentials • Prevention of user impersonation • Enforcing user authentication at all occasions • Role based access controls • Securing data in transit

Challenge Category	Challenge	Description	Threat Scenarios	Solution/security control attributes
		for predictive analytics or historical analysis. NoSQL databases only have a very thin security layer, compared to traditional RDBs		• Logging • Data tagging • Encryption
Data privacy	Scalable privacy-preserving data mining and analytics – data usage rights management	A major challenge in Big Data is that the usage can lead to and enable invasions of privacy, invasive marketing that could put Visa at risk legally if data usage rights are not effectively managed	Analysts are frequently using, merging, and combining data sets for data intelligence and often have the ability to extract private and sensitive information either maliciously or not. Knowledge of PII is a big threat	Best practices include: • Encryption of data at rest • Data segmentation and anonymization • Data metering and monitoring combinations to detect anomalies • A strict Role based [RBAC] and attribute based access control [ABAC] model
	Cryptographically enforced data centric security	Control of data access and achieving confidentiality has often been through access controls in all areas and is especially true in Big Data scenarios.	Key threats include: • Being able to identify clear text from the cipher text • Ability to process encrypted data • Access to the keys to enable access to the	Key controls include: • Using tamper resistant mechanisms to protect and separate the keys from the data • Separating and partitioning access control mechanism in conjunction with RBAC thereby separating out data access entitlements from

Challenge Category	Challenge	Description	Threat Scenarios	Solution/security control attributes
		Combining this with cryptographic protection is the 'gold standard' for data protection	data	administrative entitlements from user management entitlements
	Granular access control			
Data management	Secure data storage and transactions logs	With the exponential growth of data sets, auto-tiering of data is imperative for scalability and availability. Auto-tiering solutions do not keep track of where the data is stored, which poses new challenges to secure data storage. New mechanisms are imperative to thwart unauthorized access and maintain constant availability	• Confidentiality and integrity at the storage tier • Availability of the data • Consistency of the data provided back	• Robust encryption of data
	Granular audits			
	Data provenance			
Integrity and	End point	Many Big Data	Key threat scenarios	There is no fool proof

Challenge Category	Challenge	Description	Threat Scenarios	Solution/security control attributes
reactive security	validation and input filtering	uses in enterprise settings necessitate data accumulation from a variety of sources. A key challenge in the data load process is input validation – "ability to trust the data."	include: • Tampering of the data source • Malicious input data • Compromise of data in transmission during data load	approach for input validation. Key controls will include: • Ingestion of data from trusted and mutually authenticated sources through secure mechanisms to mitigate the threats from malicious entry and compromise of transmission
	Real-time monitoring	A key challenge is real-time monitoring of the infrastructure itself and the accesses. The Big Data platform has a greater challenge due to potentially the large volumes of false positives	Key threats include: • Ability to quickly identify anomalies from analysts' activities • Insider threats to the monitoring data • Security of the monitoring applications and the logs	The best approaches to real-time monitoring are around: • Building data metering capabilities to detect anomalies • Using a separate SIEM infrastructure away from the Data lakes to ensure integrity of the logs and preventing internal attacks on them

3.2.2 Hadoop Threat Modeling

Shown here in the overall Hadoop Threat Model that is reflective of a typical Hadoop deployment in an enterprise. The assumptions being made are the following:

- External entities (e.g. partners, customers, 3rd parties, maybe corporate access) are trying to connect to or perform a job using data present on your Hadoop cluster
- There is the standard segmentation of networks (external network, corporate network,

an application network/trust boundary, and a restricted network for data incl. Hadoop clusters)

- While not typical, another assumption being made is that all traffic is being run over a secure https channel (TLS, SSL, SFTP, etc.)

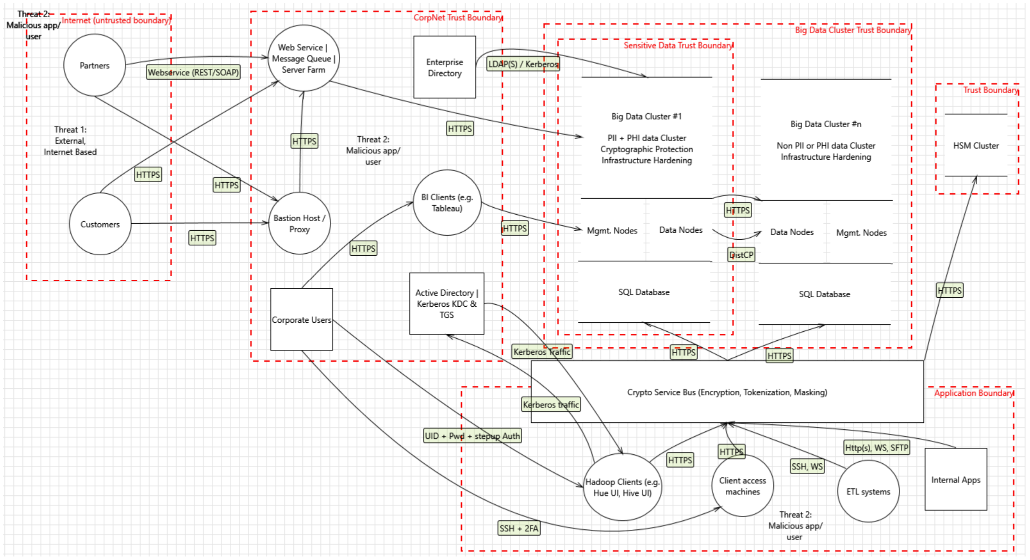
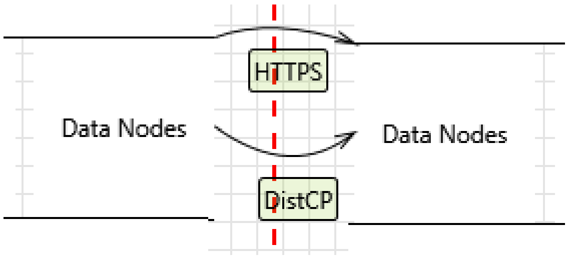


Figure 3.2 Hadoop Threat Model

The different threats are shown below:

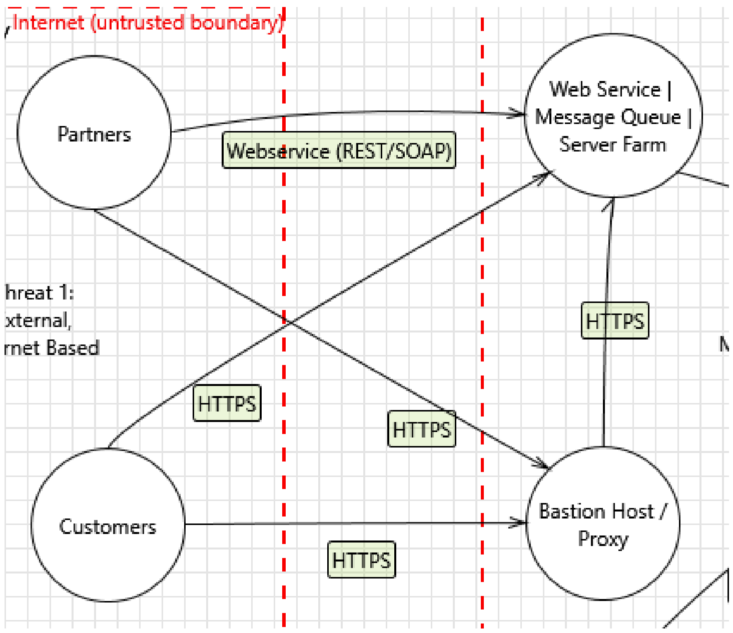
Interaction between Data Nodes:



#	Threat name	Threat Category	Description
1	Spoofing of ource and destination Data Nodes	Spoofing is when a process or entity is something other than its claimed identity. Examples include substituting a	Data Nodes may be spoofed by an attacker and this may lead to incorrect data delivered to Data Nodes. Consider using a standard

		process, a file, website or a network address.	authentication mechanism to identify the source data store.
2	Data Store Denies Data Nodes Potentially Writing Data	Repudiation threats involve an adversary denying that something happened.	Data Nodes claims that it did not write data received from an entity on the other side of the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.
3	Data Flow DistCP Is Potentially Interrupted	Denial of Service happens when the process or a datastore is not able to service incoming requests or perform up to spec.	An external agent interrupts data flowing across a trust boundary in either direction.
4	Data Store Inaccessible	Denial of Service happens when the process or a datastore is not able to service incoming requests or perform up to spec.	An external agent prevents access to a data store on the other side of the trust boundary.

Interaction with HTTPS



#	Threat name	Threat Category	Description
5	External Client Process Memory Tampered	Tampering is the act of altering the bits. Tampering with a process involves changing bits in the running process. Similarly, Tampering with a data flow involves changing bits on the wire or between two running processes.	If *Customers* is given access to memory, such as shared memory or pointers, or is given the ability to control what Web Service | Message Queue | Server Farm executes (for example, passing back a function pointer.), then Customers can tamper with Web Service | Message Queue | Server Farm. Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.
	Elevation Using Impersonation	A user subject gains increased capability or privilege by taking advantage of an implementation bug.	Web Service | Message Queue | Server Farm may be able to impersonate the context of Customers in order to gain additional privilege.
	Elevation by Changing the Execution Flow in Web Service | Message Queue | Server Farm	A user subject gains increased capability or privilege by taking advantage of an implementation bug.	An attacker may pass data into Web Service | Message Queue | Server Farm in order to change the flow of program execution within Web Service | Message Queue | Server Farm to the attacker's choosing.
	Web Service | Message Queue | Server Farm May be Subject to Elevation of Privilege Using Remote Code Execution	A user subject gains increased capability or privilege by taking advantage of an implementation bug	Customers may be able to remotely execute code for Web Service | Message Queue | Server Farm.
	Data Flow HTTPS Is Potentially Interrupted	Denial of Service happens when the process or a datastore is not able to service incoming requests or perform up to spec.	An external agent interrupts data flowing across a trust boundary in either direction.
	Potential Process Crash or Stop for Web Service | Message Queue | Server Farm	Denial of Service happens when the process or a datastore is not able to service incoming requests or perform up to spec.	Web Service | Message Queue | Server Farm crashes, halts, stops or runs slowly; in all cases violating an availability metric.

	Potential Data Repudiation by Web Service | Message Queue | Server Farm	Repudiation threats involve an adversary denying that something happened.	Web Service | Message Queue | Server Farm claims that it did not receive data from a source outside the trust boundary. Consider using logging or auditing to record the source, time, and summary of the received data.
	Spoofing the External Client (2) Process	Spoofing is when a process or entity is something other than its claimed identity. Examples include substituting a process, a file, website or a network address.	Customers may be spoofed by an attacker and this may lead to unauthorized access to Web Service | Message Queue | Server Farm. Consider using a standard authentication mechanism to identify the source process.

3.3 Hadoop Security Framework

The following section provides the target security architecture framework for Hadoop security. The core components of the proposed security framework are the following:

1. Data Management
2. Identity & Access Management
3. Data Protection & Privacy
4. Network Security
5. Infrastructure Security & Integrity

These '5 pillars' of Hadoop Security Framework are further decomposed into 21 sub-components, each of which are critical to ensuring the security and mitigating the security risk and threat vectors to the Hadoop stack. The overall security framework is shown below.

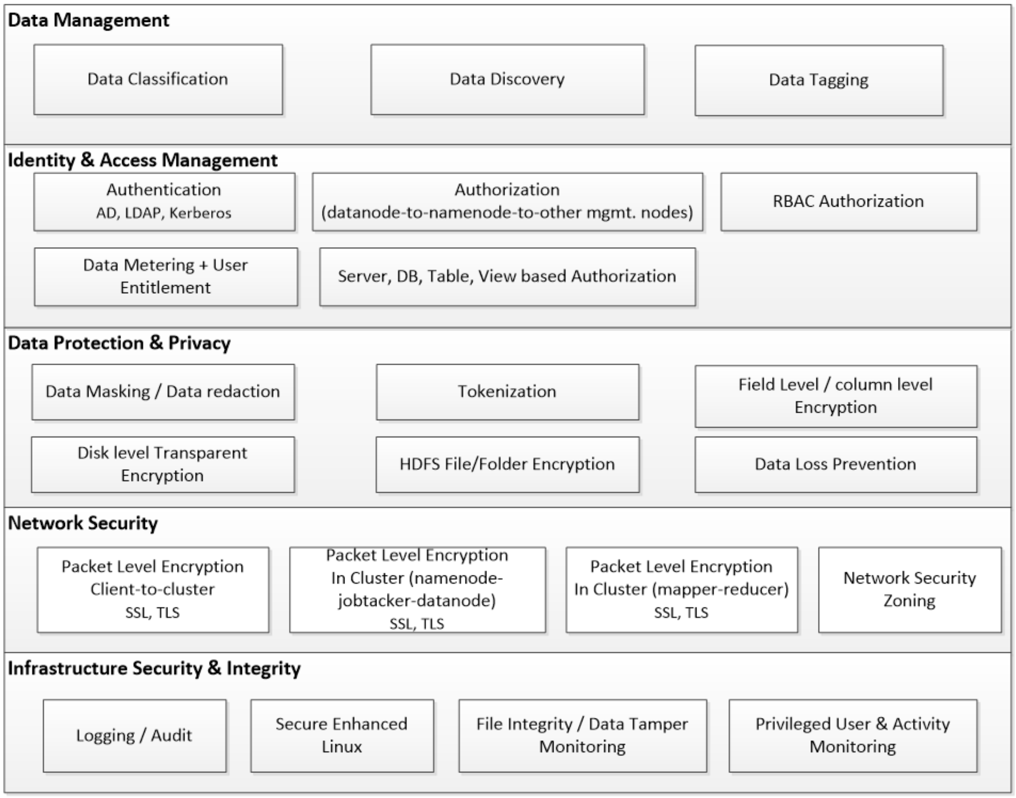

Data Management
Data Classification
Data Discovery
Data Tagging
Identity & Access Management
Authentication AD, LDAP, Kerberos
Authorization (datanode-to-namenode-to-other mgmt. nodes)
RBAC Authorization
Data Metering + User Entitlement
Server, DB, Table, View based Authorization
Data Protection & Privacy
Data Masking / Data redaction
Tokenization
Field Level / column level Encryption
Disk level Transparent Encryption
HDFS File/Folder Encryption
Data Loss Prevention
Network Security
Packet Level Encryption Client-to-cluster SSL, TLS
Packet Level Encryption In Cluster (namenode-jobtacker-datanode) SSL, TLS
Packet Level Encryption In Cluster (mapper-reducer) SSL, TLS
Network Security Zoning
Infrastructure Security & Integrity
Logging / Audit
Secure Enhanced Linux
File Integrity / Data Tamper Monitoring
Privileged User & Activity Monitoring

Figure 3.3: Hadoop Security Framework

3.3.1 Data Management

Data Management component is decomposed into three core sub-components. They are Data Classification, Data Discovery, and Data Tagging.

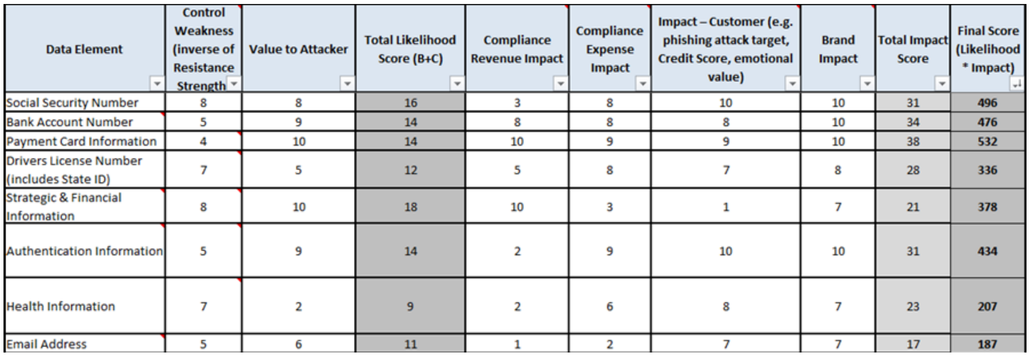
DATA CLASSIFICATION

Effective data classification is probably one of the most important activities that can in-turn lead to effective security control implementation in a Big Data platform. When organizations deal with an extremely large amount of data, aka Big Data, by clearly being able to identify what data matters, what needs cryptographic protection among others, and what fields need to be prioritized first for protection, more often than not determine the success of a security initiative on this platform.

The following are the core items that have been developed over time and can lead to a successful data classification matrix of your environment.

1. Work with your legal, privacy office, Intellectual Property, Finance, and Information Security to determine all distinct data fields. An open bucket like health data is not sufficient. This exercise encourages the reader to go beyond the symbolic policy level exercise.
2. Perform a security control assessment exercise.
   a. Determine location of data (e.g. exposed to internet, secure data zone)
   b. Determine number of users and systems with access
   c. Determine security controls (e.g. can it be protected cryptographically)
3. Determine value of the data to the attacker
   a. Is the data easy to resell on the black market?
   b. Do you have valuable Intellectual Property (e.g. a nation state looking for nuclear reactor blueprints)
4. Determine Compliance and Revenue Impact
   a. Determine breach reporting requirements for all the distinct fields
   b. Does loss of a particular data field prevent you from doing business (e.g. card holder data)
   c. Estimate re-architecting cost for current systems (e.g. buying new security products)
   d. Other costs like more frequent auditing, fines and judgements and legal expenses related to compliance
5. Determine impact to the owner of the PII data (e.g. a customer)
   a. Does the field cause phishing attacks (e.g. email) vs. just replace it (e.g. loss of a credit card)

The following figure is a sample representation of certain Personally Identifiable Data fields



Data Element	Control Weakness (inverse of Resistance Strength	Value to Attacker	Total Likelihood Score (B+C)	Compliance Revenue Impact	Compliance Expense Impact	Impact – Customer (e.g. phishing attack target, Credit Score, emotional value)	Brand Impact	Total Impact Score	Final Score (Likelihood * Impact)
Social Security Number	8	8	16	3	8	10	10	31	496
Bank Account Number	5	9	14	8	8	8	10	34	476
Payment Card Information	4	10	14	10	9	9	10	38	532
Drivers License Number (includes State ID)	7	5	12	5	8	7	8	28	336
Strategic & Financial Information	8	10	18	10	3	1	7	21	378
Authentication Information	5	9	14	2	9	10	10	31	434
Health Information	7	2	9	2	6	8	7	23	207
Email Address	5	6	11	1	2	7	7	17	187

Figure 3.4 Data Classification Matrix

3.3.2 Data Discovery

The lack of situational awareness with respect to sensitive data could leave an organization exposed to significant risks. Identifying whether sensitive data is present in Hadoop, where it is located and subsequently triggering the appropriate data protection measures, such as data masking, data redaction, tokenization or encryption is key.

- For structured data going into Hadoop, such as relational data from databases, or, for example, comma-separated values (CSV) or JavaScript Object Notation (JSON)-formatted files, the location and classification of sensitive data may already be known. In this case, the protection of those columns or fields can occur programmatically, with, for example, a labeling engine that assigns visibility labels/cell level security to those fields.
- With unstructured data, the location, count and classification of sensitive data becomes much more difficult. Data discovery, where sensitive data can be identified and located, becomes an important first step in data protection.

The following items are crucial for an effective data discovery exercise of your Big Data environment:

1. Define and validate the data structure and schema. This is all useful prep work for data protection activities later
2. Collect metrics (e.g. volume counts, unique counts etc.). For example, if a file has 1M records but it is duplicate of a single person, it is a single record vs. 1M records. This is very useful for compliance but more importantly risk management.
3. Share this insight with your Data Science teams for them to build threat models, profiles that will be useful in data exfiltration prevention scenarios.
4. If you discover sequence files, work with your application teams to move away from this data structure. Instead, leverage columnar storage formats such as Apache Parquet where possible regardless of the data processing framework, data mode, or programming language.
5. Build conditional search routines (e.g. only report on date of birth if a person’s name is found or Credit Card # + CVV or CC +zip)
6. Account for usecases where once sensitive data has been cryptographically protected (e.g. encrypted or tokenized), what is the usecase for the discovery solution.

3.3.3 Data Tagging

Understand the end-to-end data flows in your Big Data environment, especially the ingress and egress methods.

1. Identify all the data ingress methods in your Big Data cluster. These would include all manual (e.g. Hadoop admins) or automated methods (e.g. ETL jobs) or those that go through some meta-layer (e.g. copy files or create + write).
2. Knowing whether data is coming in leveraging Command Line Interface or though some Java API or through Flume or Sqoop import of if it is being SSH’d in is important.

3. Similarly, follow the data out and identify all the egress components out of your Big Data environment.
4. This includes whether reporting jobs are being run through Hive queries (e.g. through ODBC/JDBC), through Pig jobs (e.g. reading files or Hive tables or HCatalog), or exporting it out via Sqoop or copying via REST API, Hue etc. will determine your control boundaries and trust zones.
5. All of the above will also help in data discovery activity and other data access management exercises (e.g. to implement RBAC, ABAC, etc.)

3.4 Identity & Access Management

POSIX-style permissions in secure HDFS are the basis for many access controls across the Hadoop stack.

3.4.1 Threat Modeling

The following are the biggest threats around Hadoop Identity and Access Management.

Table 3.2 Identity & Access Management Threat Modeling

Threat	Description	Security Controls
Root and privileged users	• The root/privileged user can read memory in the kernel • This user can install kernel modules • Attach a debugger to any process • Kerberos keytab files on the filesystem can be accessed by this user which can allow for impersonation of other users/services	• Privileged user monitoring • Encryption at the file/disk level or the HDFS level • Moving the keytab files to a secure credential repository
Account compromise	• It is challenging to distinguish between normal users and those accounts that have been compromised	• Threat Intelligence and monitoring
Unauthorized access	• Attacker tries to gain credentials and privileges that they currently do not possess.	• Authentication • Authorization • Auditing • User entitlement

Hadoop Cluster Management security frequently involves token delegation or the negotiation of shared secrets between nodes and services as well as managing the lifetime and renewal of (delegated) tokens and shared secrets:

- Starting and authenticating services: For example, generating passwords for OS users specific to the service at install time
- Service-to-service authentication and authorization: Authentication between Hadoop

services that will not act on behalf of the user — for example, the interaction between DataNodes and NameNodes or between NodeManagers and ResourceManagers for the purpose of monitoring and cluster coordination

- User-to-service authentication: Authentication of (enterprise) users to Hadoop components and services
- Performing operations and executing queries with the user's identity: For example, through token delegations

Based on the different IAM based threats, the following is the IAM security framework proposed.

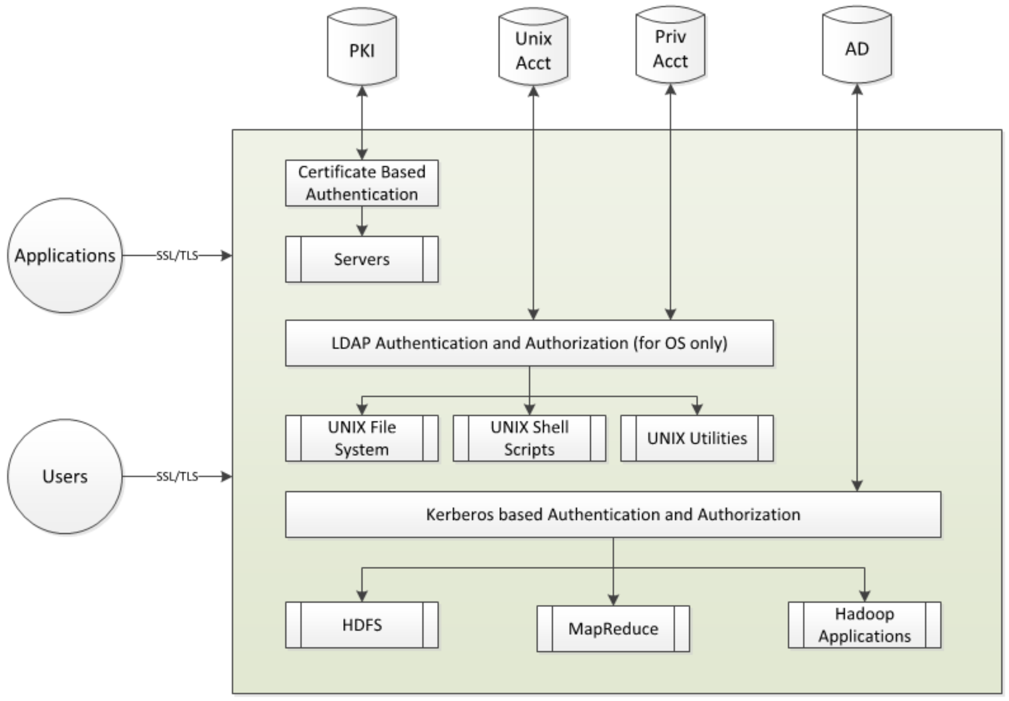


Figure 3.5: IAM Security Framework

3.4.2 *Authentication*

Authentication is the process of demonstrating that an entity is who they claim to be. In Hadoop, the primary authentication mechanisms are Kerberos and LDAP.

- Kerberos serves as the method of authenticating the Hadoop services and is the mechanism used by any agents/entities trying to connect to a Hadoop cluster.
- LDAP is another common method for authentication especially since it offers tighter

integration with many client software components. It is also attractive option when there are external clients trying to authenticate to the Hadoop cluster.

3.4.3 Authorization

Authorization is the validation process of a particular entity's access and verifying whether they have access to a particular resource and can perform any action on it.

- The best method for authorization in Hadoop is file and directory permissions in HDFS
- Other Hadoop controls (e.g. Apache Sentry) can handle authorization at higher levels of abstraction that HDFS
- SAML is also used to provide authorization especially when clients are used and where the role is passed in the SAML assertion for the user being authenticated.

3.4.4 Access Control

Hadoop security incorporates strong access control: Label security and cell-level security are common in Hive and HBase, and they were part of Accumolo even at the time of its inception. Label security and cell-level security are usually built into the application or the data rather than keeping it outside in dedicated ACLs, policy files or applications.

Label security: Label security and cell-level security make RBAC part of the database and do away with the need for clunky text-based configuration files.

- Label security is available for the key/value stores (e.g. OSS HBase) and Visa can benefit from it only when there is a way to ingest the data into, for example, HBase.
- Use cases and access patterns of key/value stores and SQL-like data stores are not interchangeable, and the security implications differ.
- Although HBase API security may focus on securing the RESTful API, Hive access security may want to prevent typical SQL attacks.

3.4.5 User Entitlement + Data Metering

Provide users access to data by centrally managing access policies.

- It is important to tie policy to data and not to the access method
- Leverage Attribute Based Access Control (ABAC) and protect data based on tags that move with the data through lineage; permissions decisions can leverage the user, environment (e.g. location), and data attributes.
- Perform data metering by restricting access to data once a normal threshold (as determined by access models + machine learning algorithms) is passed for a particular user/application.

3.4.6 RBAC Authorization

Deliver fine-grained authorization through Role Based Access Control (RBAC).

- Manage data access by role (and not user)
- Determine relationships between users & roles through groups. Leverage AD/LDAP

group membership and enforce rules across all data access paths

3.4.7 HDFS Security

Hadoop security begins with securing HDFS. The security model for HDFS is based on the POSIX security model. In this model, access permissions are assigned to groups and users. Each file or directory has read and write permissions for its owner, group, or all users. The user identity is determined by the host operating system when Hadoop is in "simple" mode. If Kerberos support is turned on, the identity comes from the Kerberos credentials that may also be shared with LDAP. Note that there are no execute permissions in the HDFS security model because files cannot be executed; MapReduce programs execute from outside the HDFS file system.

The way that access is resolved is very simple. When a user tries to access a file, the NameNode uses native (typically Linux) commands to get the user's groups either through a JNI interface or by actually shelling out and executing the bash command *groups*. Thus the NameNode machine needs to know the complete user and group set when using the simple model. The user and groups for a file on HDFS are stored in the NameNode as strings.

When you start the NameNode, the user who starts it becomes the superuser for all of HDFS and will have complete access to all of the files. That user doesn't need to be root on the NameNode (and really shouldn't be for security purposes).

During Hadoop installation, a common strategy is to create several different users and groups for Hadoop. The hdfs user becomes the superuser for HDFS while the yarn user and mapred users own resource management and MapReduce process logging, respectively. Each user also has a group associated with it and these users own the corresponding file system resources on local disks across the cluster. Finally, a hadoop group provides for general access on HDFS but without a corresponding user. All Hadoop users should be in the hadoop group. Table shows a standard configuration:

Table 3.4 File System and HDFS directory ownership for standard configuration

User	Programs	File System Directories	Logging	HDFS Directories
hdfs	NameNode, DataNode, Secondary NameNode	dfs.namenode.name.dir dfs.datanode.data.dir	HDFS_LOG_DIR	/ (ALL)
yarn	ResourceManager, NodeManager	yarn.nodemanager.local-dirs yarn.nodemanager.log-dirs container-executor conf/container-executor.cfg	YARN_LOG_DIR	yarn.nodemanager.remote-app-log-dir
mapred	Job History Server		MAPRED_LOG_DIR	mapreduce.jobhistory.intermediate-done-dir

				mapreduce.jobhistory.done-dir

In addition to these basic services, each additional service like Hive, Tez, Pig, etc. should have its own user and will also leverage the hadoop group. All ordinary user of the Hadoop cluster will also need to be added to the hadoop group.

Finally, in Hadoop there is the idea of a supergroup that does not necessarily have a group affiliation on the file system. When the HDFS superuser (whoever starts the NameNode) initially formats HDFS, the group assignment for the root of HDFS will be assigned to *supergroup*. A new user who is a member of the hadoop group will not be able to create directories or files at the /-level of the file system. They probably shouldn't, either. The general best practice is to create user directories that they can access. However, you can also create different sets of privileged users, assigning them to a new group and then set the supergroup to that group. For example, let's say that you have two users who need to have HDFS superuser permissions. You can create a new Unix/Linux group, hdfssuper, and add the users to it:

```
# groupadd hdfssuper
# usermod -a -G hdfssuper myuser1
# usermod -a -G hdfssuper myuser2
```

Next, you will change the property dfs.permissions.supergroup int etc/hadoop/hdfs-site.xml to the new hdfssuper group that you created and restart the NameNode. Then both myuser1 and myuser2 will be able to act as the superuser. The files they create will still be labeled as being supergroup rather than their file system groups, but the *supergroup* designation will now be mapped to hdfssuper.

From a MapReduce programming perspective, the HDFS security model carries over just like in Unix and Linux. If you run a MapReduce program as a certain user, that program will only be able to read and write files, create directories, and so forth where it has permissions to do so. Since you may execute the program from a computer that is not even part of the Hadoop cluster, your user and groups system must be shared across all of the machines that are involved in the Hadoop environment. Direct HDFS programming operations using the DFS API in Hadoop will carry the user credentials as well.

3.4.8 ACLs and Security

Access Control Lists (ACLs) are now supported in Hadoop as of Apache Hadoop 2.4. ACLs are an extension to the basic user and group security model inherited from POSIX. ACLs fix a fundamental problem with the POSIX model: access patterns may not map cleanly to organizational structures.

Let's say that user jane owns a file that contains a list of customers for a sales territory. Jane wants to let others in that territory read the file but reserves the modify permissions to herself. That is easily done in the file systems permissions model by setting read and write access to jane while creating a group westcoast that includes the others sales personnel in her territory. The group permissions are then set to read-only.

Now things change and Jane needs to delegate change permissions to three new regional managers while still keeping read-only for the rest of the sales organization. There is really no way to do this because you can't have two groups have separate permissions for the file, and you don't want everyone to be able to read the file. Traditional fixes to this problem were to create a synthetic user who Jane and her managers could su to when needed, but this seems crude and unnecessarily complex.

ACLs solve this dilemma by supporting a list of permissions that are considered in order. On Hadoop, you can turn on ACL support with the following property in hdfs-site.xml:

```
<property>
    <name>dfs.namenode.acls.enabled</name>
    <value>true</value>
</property>
```

With this property enabled you can now set and retrieve the ACLs on a file. You can set the user, group and all access, but you can also deny access to a single user or a group, or can enable access to others.

The commands used to retrieve and view the ACL for a file or directory is getacl:

```
$ hdfs dfs -getfacl /sales-data.txt 14/12/24 13:40:46
# file: /sales-data.txt
# owner: mark
# group: supergroup
user::rw-
group::r—
other::---
```

In this example, the basic file system permissions are the only access patterns enabled. These are inherited from the basic security system. But we can easily add to the list. For instance, if we want to allow a group called sales1 to modify the file, we just need to add a permission for that group to have write access:

```
$ hdfs dfs -setfacl -m group:sales1:rw- /sales-data.txt
```

Note the –m for the ACL specification and the requirement that each bit (read, write, and execute) be specified or given as a – to indicate not set or remove permission. Now if we get the ACLs again:

```
hdfs dfs -getfacl /sales-data.txt
# file: /sales-data.txt
# owner: mark
# group: supergroup
user::rw-
group::r—
group:sales1:rw-
mask::rw-
other::---
```

The sales1 group has now been given read and write permission for the file. In addition, the keyword “default” can be prefixed to the ACL specification in the setfacl call. Default then applies to subdirectories and files in those subdirectories when working with changing the ACLs of files. Also, the setfacl and getfacl calls support the –R flag to provide recursive operations just like chmod and chgrp.

Taken together, the basic permissions system combined with the ACL overlay provides a thorough security and access management system that is comparable to Unix/Linux and Windows systems for security. What we’ve discussed so far does not integrate with other systems for user authentication and on-the-wire encryption. To get those capabilities, we need LDAP and Kerberos, the subjects of the next sections.

3.4.9 LDAP for Hadoop

Security and permissions in Linux are distributed in that the groups and users are resolved on each computer in the network subdomain and Hadoop cluster. This means that in order to create a user who can access all the machines in a large cluster, scripts or tools are needed that visit each computer and execute a shell command to add the user.

LDAP provides a centralized directory service for resolving user identities and for other purposes like organizational structures. For Hadoop, LDAP support can replace the mappings of user identities to permissions, even using LDAP capabilities that are shared with Active Directory on Windows computers.

Configuring LDAP for Hadoop can be challenging. If your organization has existing LDAP servers, the users and groups need to be entered into the directory. This is done using file that specifies the user identity, their organization unit, and other useful facts. The file format is the LDAP Data Interchange Format (LDIF) and entries look like this:

```
dn: cn=hdfs,ou=services,dc=my-company,dc=com
objectclass:top
objectclass: applicationProcess
objectclass:simpleSecurityObject
cn: hdfs
userPassword:hdfs-password
```

This file is then imported into the LDAP database using a tool like ldapadd on Linux using OpenLDAP:

```
ldapadd -f hadoop.ldif -D cn=manager,dc=hadoop,dc=my-company,dc=com -w hadoop
```

After setting up LDAP to provide the authorization, the next step is to set up Hadoop to use LDAP to do the authorization. You modify core-site.xml to change Hadoop’s method, adding a range of parameters to the file and restarting:

```
<property>
  <name>hadoop.security.group.mapping</name>
  <value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.user</name>
  <value>cn=Manager,dc=my-company,dc=com</value>
```

```
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.password</name>
  <value>ldappassword</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.url</name>
  <value>ldap://localhost:389/dc=my-company,dc=com</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.base</name>
  <value></value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.user</name>
  <value>(&amp;(|(objectclass=person)(objectclass=applicationProcess))(cn={0}))
  </value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.group</name>
  <value>(objectclass=groupOfNames)</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.member</name>
  <value>member</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.attr.group.name</name>
  <value>cn</value>
</property>
```

The critical components of this configuration should be familiar to anyone who deals with LDAP regularly. The most important property is `hadoop.security.group.mapping.ldap.url` that provides the URL of the LDAP server. In a typical Hadoop deployment that is secured with LDAP, you have to configure the firewall to communicate with the LDAP server.

3.4.10 Kerberos and Hadoop

Kerberos is a security system that not only authenticates but also provides secure communications between clients and servers. Kerberos is a proven system for security that is widely used. Windows security is based on Kerberos, for example. The Kerberos protocol is designed to avoid some of the challenges with traditional security mechanisms, but it was also designed primarily for two-tiered architectures where a client authenticates to a server and then is allowed to communicate with the server for a session. Hadoop has many different programs working together, and so authenticating to the ResourceManager to run a job also needs to support carrying that authorization through to the NameNode for access to HDFS resources.

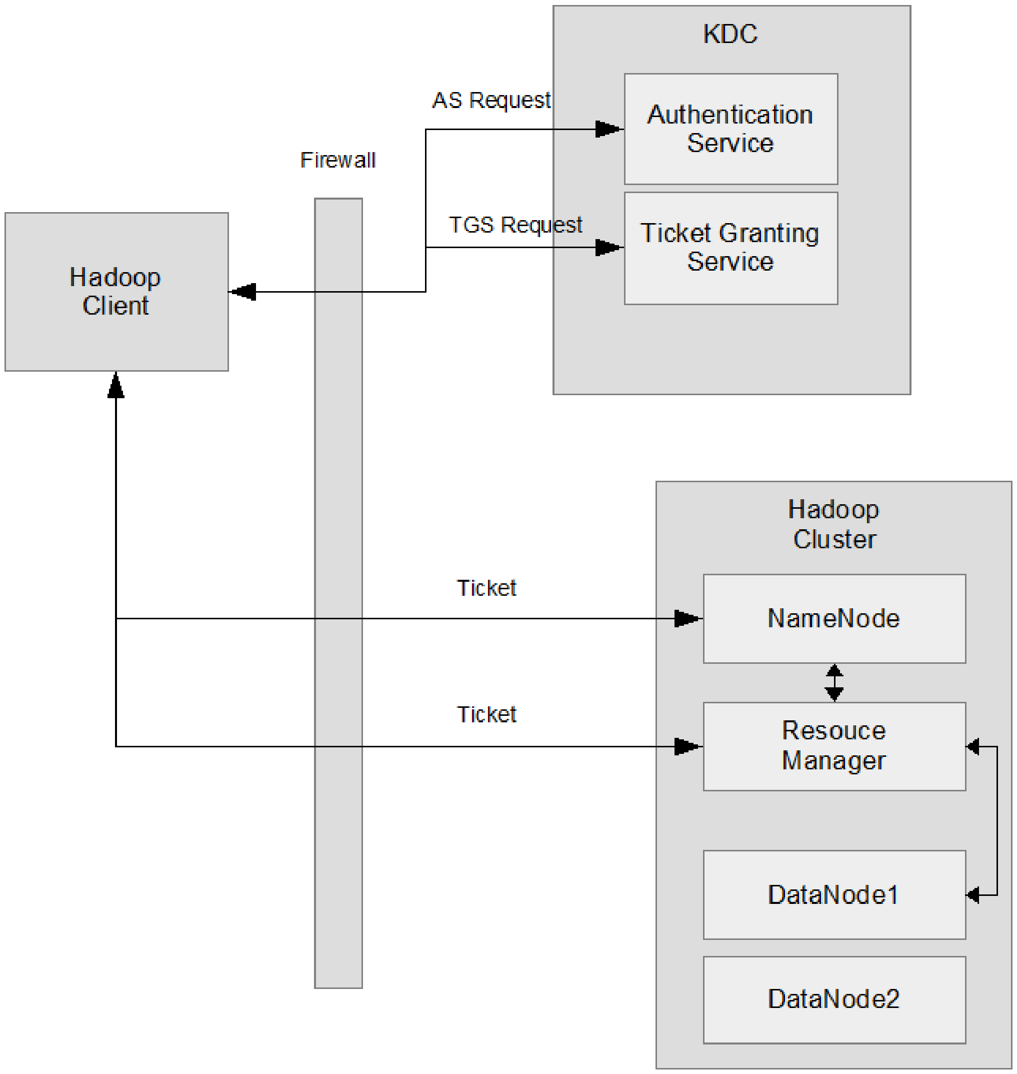


Figure 3.6: Hadoop client interacting with Kerberos-secured Hadoop. The KDC provides authentication and tickets for access to the Hadoop cluster.

With Kerberos, the client program does not transmit the user password to the Kerberos Authentication Server (AS) but instead creates a one-way hash out of it using the agreed to protocol. The client program does transmit the user identifier to the AS without bothering to encrypt it or the communications channel. The AS then encrypts a session key (if the user exists) using the user password as the key and returns that key along with an encrypted additional communication session key that will be used for the next phase of communications between the client and another Kerberos server called the Ticket Granting Server (TGS). If the client program cannot decrypt the session key, the user entered the wrong password. The protocol then continues to issue "tickets" that authenticate to the individual services from the client. Both the AS and the TGS are part of the Key Distribution Center (KDC). **Error! Reference source not found.** shows the interactions between clients and the different parts of the KDC, as well as with ticketed access to Hadoop.

In Hadoop, additional tickets need to be proxied between the individual programs needed to execute actions over Hadoop, but this generally happens transparently in the background when running in secure mode. Configuring secure mode is a bit complex, but perhaps less daunting than managing the LDAP system without special tools. The major Hadoop vendors have also improved the installation and maintenance of Kerberos with Hadoop. Getting started with Kerberos and Apache requires installing Kerberos and configuring it, then configuring Hadoop to use Kerberos.

3.4.11 Getting and Installing Kerberos

All of the major Linux distributions have Kerberos available for installation. Before starting installation, you should insure that your cluster has fully-qualified domain names (FQDNs) for each node in the cluster. That means assigning each a resolvable host and domain as well as providing some kind of DNS services for them. For our example installation, we will use CentOS 6 running in a virtualization environment. By default, the system just installs as localhost and needs a FQDN for Kerberos to work with. To change this, you modify /etc/hosts as root and set HOSTNAME /etc/sysconfig/network. You can either reboot or restart networking to have the change take effect. For our example, we have assigned hadoop.example.com as the FQDN.

You will need to install several components of Kerberos as root:

```
# yum install krb5-libs krb5-server krb5-workstation
```

After installation, you create the initial configuration database for Kerberos:

```
# kdb5_util create -s
```

Then you will need to modify the configuration files to make your hostname the primary "realm" of Kerberos system. In Kerberos, credentials are associated with a principal, an instance, and a realm. For most Linux applications—including Hadoop—the principal will be the user name and the realm will be mapped to the FQDN, hadoop.example.com in our case.

The Kerberos configuration files are divided into two parts: /etc/krb5.conf and /var/kerberos/krb5kdc. In /etc/krb5.conf are the mappings between the realm and the hostname. In this file, realms are always in uppercase while domain names are in lower:

```
[logging]
default = FILE:/var/log/krb5libs.log
kdc = FILE:/var/log/krb5kdc.log
admin_server = FILE:/var/log/kadmind.log
```

```
[libdefaults]
default_realm = EXAMPLE.COM
dns_lookup_realm = false
dns_lookup_kdc = false
ticket_lifetime = 24h
renew_lifetime = 7d
forwardable = true

[realms]
EXAMPLE.COM = {
     kdc = hadoop.example.com
     admin_server = hadoop.example.com
}

[domain_realm]
hadoop.example.com = EXAMPLE.COM
```

Note the mappings both from realm to servers (The KDC is the Key Distribution Center and comprises the Authentication and Ticket Granting servers).

Next, /var/kerberos/krb5kdc/kadm.acl needs to be modified to map the admin principal to the correct realm:

```
/admin@EXAMPLE.COM     *
```

Once completed, start the Kerberos daemons:

```
# /sbin/service krb5kdc start
# /sbin/service kadmin start
```

At this point, you need to create the principals for your different user identities. You can do that with the kadmin.local command-line client, like this:

```
# kadmin.local
addprinc root/admin
addprinc hdfs/admin
```

Each time you will need to provide the admin password that you originally created when you created the Kerberos database. You will also need to provide new passwords for the users. It is possible to combine the Kerberos password authentication with LDAP but we will not cover that here.

You can test the ticket granting service from the command line using kinit. After getting a ticket, you can list your tickets using klist. The final step is in configuring Hadoop to use Kerberos. This involves modifying Hadoop etc/hadoop/core-site.xml for all the nodes in the cluster as well as generating “keytab” files for each host and each service on each host. A keytab file contains the principal and encrypted keys for a given service. This can be time consuming and it is helpful to create some automation to help with the task.

For Hadoop, there need to be keytab files on each cluster host for the different principals needed for the configuration. A typical set of principals is shown in Table.

Table 2: Kerberos principals for Hadoop configurations

Host	Principal
NameNode	hdfs
Secondary NameNode	hdfs
DataNode	Hdfs, mapred
ResourceManager	yarn
HTTP services	HTTP

You can create these principals and generate the keytab files using Kerberos utilities:

```
# kadmin.local
kadmin.local: addprinc -randkey hdfs/hadoop.example.com@EXAMPLE.COM

WARNING: no policy specified for hdfs/hadoop.example.com@EXAMPLE.COM; defaulting to
      no policy
Principal "hdfs/hadoop.example.com@EXAMPLE.COM" created.

kadmin.local:  xst -norandkey -k hdfs.keytab hdfs/hadoop.example.com

Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type aes256-cts-
hmac-sha1-96 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type aes128-cts-
hmac-sha1-96 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type des3-cbc-
sha1 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type arcfour-hmac
added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type des-hmac-
sha1 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type des-cbc-md5
added to keytab WRFILE:hdfs.keytab.
```

Once all of the keytab files and principals have been created, they should be moved to the etc/hadoop directory and their permissions set to prevent reading by any other user. Finally, it is a simple matter to set Hadoop to secure mode and restart the cluster. Add the following to etc/hadoop/core-site.xml on every node:

```
<property>
  <name>hadoop.security.authentication</name>
  <value>kerberos</value>
</property>

<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
```

Finally, YARN and HDFS security need to be configured in etc/hadoop/hdfs-site.xml and etc/hadoop/yarn-site.xml. The following parameters are for HDFS and DataNode security:

```
<!-- HDFS Kerberos config -->
<property>
  <name>dfs.block.access.token.enable</name>
  <value>true</value>
</property>

<!-- NameNode Kerberos config -->
<property>
  <name>dfs.namenode.keytab.file</name>
  <value>/etc/hadoop/ hdfs.keytab</value> <!-- path to the HDFS keytab -->
</property>
<property>
  <name>dfs.namenode.kerberos.principal</name>
  <value>hdfs/hadoop.example.com@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/hadoop.example.com@EXAMPLE.COM</value>
</property>

<!-- Secondary NameNode Kerberos config -->
<property>
  <name>dfs.secondary.namenode.keytab.file</name>
  <value>/etc/hadoop/hdfs.keytab</value> <!-- path to the HDFS keytab -->
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.principal</name>
  <value>hdfs/hadoop.example.com@EXAMPLE.COM </value>
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/hadoop.example.com@EXAMPLE.COM</value>
</property>

<!-- DataNode Kerberos config -->
<property>
  <name>dfs.datanode.data.dir.perm</name>
  <value>700</value>
</property>
<property>
  <name>dfs.datanode.address</name>
  <value>0.0.0.0:1004</value>
</property>
<property>
  <name>dfs.datanode.http.address</name>
  <value>0.0.0.0:1006</value>
</property>
<property>
  <name>dfs.datanode.keytab.file</name>
  <value>/etc/hadoop/hdfs.keytab</value>
</property>
<property>
  <name>dfs.datanode.kerberos.principal</name>
  <value>hdfs/hadoop.example.com@EXAMPLE.COM</value>
</property>
```

And likewise for YARN:

```
<!-- ResourceManager Kerberos config -->
<property>
  <name>yarn.resourcemanager.keytab</name>
  <value>/etc/hadoop/yarn.keytab</value>
</property>
<property>
  <name>yarn.resourcemanager.principal</name>
  <value>yarn/hadoop.example.com@EXAMPLE.COM</value>
</property>

<!-- NodeManager Kerberos config -->
<property>
  <name>yarn.nodemanager.keytab</name>
  <value>/etc/hadoop/yarn.keytab</value>
</property>
<property>
  <name>yarn.nodemanager.principal</name>
  <value>yarn/hadoop.example.com@EXAMPLE.COM </value>
</property>
<property>
  <name>yarn.nodemanager.container-executor.class</name>
  <value>org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor</value>
</property>
<property>
  <name>yarn.nodemanager.linux-container-executor.group</name>
  <value>yarn</value>
</property>
```

Finally, additional configuration is needed for the mapred principal in etc/hadoop/mapred-site.xml:

```
<!-- MapReduce Job History Server Kerberos config -->
<property>
  <name>mapreduce.jobhistory.address</name>
  <value>hadoop.example.com:10020</value>
</property>
<property>
  <name>mapreduce.jobhistory.keytab</name>
  <value>/etc/hadoop/mapred.keytab</value>
</property>
<property>
  <name>mapreduce.jobhistory.principal</name>
  <value>mapred/hadoop.example.com@EXAMPLE.COM</value>
</property>
```

There is also one final configuration requirement for etc/hadoop/container-executor.cfg. Set the container executor group to yarn and add the users' hdfs, mapred, yarn, etc. to the banned users. This configuration file governs what users can execute containers and only members of the yarn group. You should also set the min.user.id to 500 if you are working on CentOS.

After all that work, you can restart your cluster and start testing functionality. Do not forget to check out your log files for both Hadoop and Kerberos (/var/log/krb5kdc.log) if things are working correctly.

3.5 Data Protection & Privacy

The majority of the Hadoop distributions and vendor add-ons package either data-at-rest encryption at a block or (whole) file level. Application level cryptographic protection (like field-level/column-level encryption, data tokenization, and data redaction/masking provide the next level of security needed.

- Full-disk encryption (FDE) can also be OS-native disk encryption, such as dm-crypt or BitLocker
- The implementation of FDE in a running Hadoop cluster is cumbersome and does not provide defense in depth for sensitive data. Gaps come from data-at-rest access by the root user or authorized users or sudoers
- The key requirement for data-at-rest encryption for technology products is compatibility with Key Management Interoperability Protocol (KMIP) standard and ability to use Hardware Security Modules (HSMs). Usage of Java KeyStore (JKS) can be acceptable in the short-term.
- While encryption at the field/element level can offer security granularity and audit tracking capabilities, it comes at the expense of requiring manual intervention to determine the fields that require encryption and where and how to enable authorized decryption.

3.5.1 Application Level Cryptography (Tokenization, field-level encryption)

While encryption at the field/element level can offer security granularity and audit tracking capabilities, it comes at the expense of requiring manual intervention to determine the fields that require encryption and where and how to enable authorized decryption.

3.5.2 Transparent Encryption (disk / HDFS layer)

Full Disk Encryption (FDE) prevents access via the storage medium. File encryption can also guard against (privileged) access at the node's operating-system level.

- In case you need to store and process sensitive or regulated data in Hadoop, data-at-rest encryption protects your organization’s sensitive data and keeps at least the disks out of audit scope.
- In larger Hadoop clusters, disks often need to be removed from the cluster and replaced. Disk Level transparent encryption ensures that no human-readable residual data remains when data is removed or when disks are decommissioned.
- Full-disk encryption (FDE) can also be OS-native disk encryption, such as dm-crypt

3.5.3 Data Masking/ Data Redaction

Data masking or data redaction before load in the typical ETL process de-identifies personally identifiable information (PII) data before load. Therefore, no sensitive data is stored in Hadoop, keeping the Hadoop Cluster potentially out of (audit) scope.

- This may be performed in batch or real time and can be achieved with a variety of designs, including the use of static and dynamic data masking tools, as well as through data services.

3.6 Network Security

The Network Security layer is decomposed into four sub-components. They are data protection in-transit and network zoning + authorization components.

3.6.1 Threat Model

Traditionally, the corporate perimeter was considered to be a trusted network. However, the security landscape has completely changed where it is never safe to trust any network communication. We should make the assumption that the attacker exists within the corporate network and must proceed accordingly with the mindset.

Table 3.3 Network Security Threat Model

Threat	Description	Controls
Service Denial	There are no native safeguards built into core Hadoop around service denial. Attacks could include the standard Denial of Service or DDoS attacks. Advanced attacks include flooding the cluster with jobs that consume all the maps allocated or consumes all the resources in the cluster.	Load balancers Rate throttling agents Network DoS protection mechanisms
Interception of communications	Network packets can be dropped, spoofed, read, or modified when in transit.	Encrypt data in transit using Transport Layer Security (TLS) or SASL encryption.

3.6.2 Data Protection In-Transit

Secure communications are required for HDFS to protect data-in-transit. There are multiple threat scenarios that in turn mandate the necessity for https and prevent information disclosure or elevation of privilege threat categories. Using the TLS protocol (which is now available in all Hadoop distributions) to authenticate and ensure privacy of communications between nodes, name servers, and applications.

- An attacker can gain unauthorized access to data by intercepting communications to Hadoop consoles.
- This could include communication between NameNodes and DataNodes that are in the clear back to the Hadoop clients and in turn can result in credentials/data to be sniffed.
- Tokens that are granted to the user post-Kerberos authentication can also be sniffed and can be used to impersonate users on the NameNode.

Following are the controls that when implemented in a Big Data cluster can ensure properties of data confidentiality.

1. Packet level encryption using TLS from the client to Hadoop cluster

2. Packet level encryption using TLS within the cluster itself. This includes using https between NameNode to Job Tracker to DataNode.
3. Packet level encryption using TLS in the cluster (e.g. mapper-reducer)
4. Use LDAP over SSL (LDAPS) rather than LDAP when communicating with the corporate enterprise directories to prevent sniffing attacks.
5. Allow your admins to configure and enable encrypted shuffle and TLS/https for HDFS, MapReduce, YARN, HBase UIs etc.

3.6.3 Network Security Zoning

The Hadoop clusters must be segmented into points of delivery (PODs) with chokepoints such as Top of Rack (ToR) switches where network Access Control Lists (ACLs) limit the allowed traffic to approved levels.

- End users must not be able to connect to the individual data nodes, but to the name nodes only.
- The Apache Knox gateway for example, provides the capability to control traffic in and out of Hadoop at the per-service-level granularity.
- A basic firewall that should allow access only to the Hadoop NameNode, or, where sufficient, to an Apache Knox gateway. Clients will never need to communicate directly with, for example, a DataNode.

3.7 Infrastructure Security & Integrity

The Infrastructure Security & Integrity layer is decomposed into four core sub-components. They are Logging/Audit, Secure Enhanced Linux (SELinux), File Integrity + Data Tamper Monitoring, and Privileged User and Activity Monitoring.

3.7.1 Threat Model Development

It may sound obvious, but if your Hadoop cluster is not configured in a secure manner, then your environment is subject to higher threat and risk exposure.

3.7.2 Logging / Audit

All system/ecosystem changes unique to Hadoop clusters need to be audited with the audit logs being protected. Examples include:

- Addition/deletion of data and management nodes
- Changes in management node states including job tracker nodes, name nodes
- Pre-shared secrets or certificates that are rolled out when the initial package of the Hadoop distribution or of the security solution is pushed to the node prevent the addition of unauthorized cluster nodes.

When data is not limited to one of the core Hadoop components, Hadoop data security ends up having many moving parts and high percentage of fragmentation. Consequently, there results a sprawl of metadata and audit logs across all fragments.

In a typical enterprise, the DBAs are typically leveraged to place the security responsibility at the table, row, column, or cell level and while the configuration of file systems and system administrators, and the Security Access Control team is usually accountable for the more granular file level permissions.

Yet, in Hadoop, POSIX-style HDFS permissions are frequently important for data security or are at times the only means to enforce data security at all. This leads to questions concerning the manageability of Hadoop security. Technologies recommendations to address data fragmentation:

- Apache Falcon is an incubating Apache OSS project that focuses on data management. It provides graphical data lineage and actively controls the data life cycle. Metadata is retrieved and mashed up from wherever the Hadoop application stores it.
- Cloudera Navigator is a proprietary tool and GUI that is part of Cloudera's Distribution Including Apache Hadoop (CDH) distribution. CDH Navigator is a tool to address log sprawl, lineage and some aspects of data discovery. Metadata is retrieved and mashed up from wherever the Hadoop application stores it.
- Zettaset Orchestrator is a product for harnessing the overall fragmentation of Hadoop security with a proprietary combined GUI and workflow. Zettaset has its own metadata repository where metadata from all Hadoop components is collected and stored.

3.7.3 Secure Enhanced Linux (SELinux)

SELinux was created by the United States National Security Agency (NSA) as a set of patches to the Linux Kernel using Linux Security Modules (LSM). It was eventually released by the NSA under the GPL license and has been adopted by the upstream Linux kernel.

SELinux is an example of a Mandatory Access Control (MAC) for Linux. The SELinux policy achieves this objective through process separation and process isolation. Process separation is achieved by confining processes to individual SELinux domains. Process isolation is achieved by granting process domains only the permissions that they require to run properly.

SELinux can also be classified as a labeling system for your platform where every process/file/directory/object has a label. The SELinux policy rules then define how such processes access the objects via labels which are then enforced by the Linux Kernel. The benefits of using SELinux are the following:

1. **Enhanced access manager**: SELinux policies can govern which entities whether applications or privileged users for example are entitled to initiate services on the Linux platform running Hadoop. Such services could include Apache, Tomcat, LDAP, etc. without requiring privilege elevation (e.g. become root)
2. Type enforcement using SELinux: SELinux can help prevent takeover of an entire system due to the compromise of a single application.

a. Can protect kernel modules, device drivers, system configuration files (Web, FTP, DNS etc.) using the SELinux security policy from unauthorized access and tampering.

b. By enforcing the default policy of least privilege, it protects privileged user environments.

3. Identity-based access control: Systems on Linux/Hadoop perform discretionary based access control. With SELinux, the security policy governs all permissions to all resources. This defines which users and programs can access any processes and objects (; i.e. files) on the system.

   a. In SE Linux, the all-powerful user root does not apply. Instead, the defining principle is to grant least privilege, by default.

USING SELINUX ON HADOOP

Historically Hadoop and other Big Data platforms built on top of Linux and UNIX systems have had discretionary access control. What this means for example is that a privileged user like root is omnipotent.

- By enforcing and configuring SELinux on your Big Data environment, through MAC, there is policy that is administratively set and fixed.
- Even if a user changes any settings on their home directory, the policy prevents another user or process from accessing it.
- A sample policy for example that can be implemented is to make library files executable but not writable or vice-versa. Jobs can write to /tmp location but not be able to execute anything in there. This is a great way to prevent command injection attacks among others.
- With policies configured, even if someone who is a sysadmin or a malicious user is able to gain access to root using SSH or some other attack vector, they may be able to read and write a lot of stuff. However, they will not be able to execute anything incl. potentially any data exfiltration methods.

The general recommendation is to run SELinux is permissive mode with regular workloads on your cluster, reflecting typical usage, including using any tools. The warnings generated can then be used to define the SELinux policy which after tuning can be deployed in a ‘targeted enforcement’ mode.

SAMPLE IMPLEMENTATION OF SELINUX POLICY

One of the most popular Hadoop implementations is Cloudera Hadoop (CDH). This sample SELinux policy is done using CDH. However, it should be abstract enough and applicable for other Hadoop distributions as well. The policy below has been developed from scratch. SELinux policy does exist for open source Hadoop in RHEL targeted policy, as well as upstream Reference Policy. Also, this section does not dive into determining a good architecture of SELinux labelling of files on the system. Labelling will need to be done for Cloudera configuration files, libraries, executables, and data stores to complete the secure posture.

Within Cloudera Hadoop, this sample is targeted against three (3) applications for SELinux policy - **/usr/sbin/cmf-server, /usr/sbin/cmf-agent, and /usr/bin/kinit**.

1. The cmf-server and cmf-agent executables were running in the initrc_t domain, which is the generic domain RHEL targeted policy will run daemons in that do not have specific SELinux policy. By labeling the /etc/init.d/cloudera-scm-server file as cloudera_server_exec_t, and adding init_daemon_domain(cloudera_server_t, cloudera_server_exect_t), we can start the cmf-server process in the cloudera_server_t domain. By confining the cmf-server process to this domain, we can restrict the set of SELinux permissions to those granted to the cloudera_server_t domain, and no longer to the permissions of the initrc_t domain (which is a highly privileged domain).
2. By labeling the /etc/init.d/cloudera-scm-agent file as cloudera_agent_exec_t, and adding init_daemon_domain(cloudera_agent_t, cloudera_agent_exec_t), we can start the cmf-agent process in the cloudera_agent_t domain. By confining the cmf-agent process to this domain, we restrict the set of SELinux permissions to those granted to the cloudera_agent_t domain, and no longer to the permissions of the initrc_t domain (which is a highly privileged domain).
3. By labeling the /usr/bin/kinit file as kinit_exec_t, and adding a domain transition from the inetd_child_t domain. Make this a proper login domain such as unconfined_t, sysadm_t, or user_t, after which we can restrict the set of permissions to those granted to the kinit_t domain. As for labeling files in home directories for which other users are not allowed to read, strict user policy will need to be enforced, with separation either through roles or through MLS/MCS.

CLOUDERA.FC

```
/etc/init\.d/cloudera-scm-server
gen_context(system_u:object_r:cloudera_server_exec_t,s0)
/etc/init\.d/cloudera-scm-agent
gen_context(system_u:object_r:cloudera_agent_exec_t,s0)
/opt/cloudera(/.*)?
gen_context(system_u:object_r:cloudera_opt_t,s0)
/usr/lib64/cmf(/.*)?
gen_context(system_u:object_r:cloudera_lib_t,s0)
/hadoop-[a-z](/.*)?
gen_context(system_u:object_r:hadoop_data_t,s0)
/usr/bin/kinit  --
gen_context(system_u:object_r:kinit_exec_t,s0)
```

CLOUDERA.TE

```
policy_module(cloudera, 1.0.0)

########################################
#
# Declarations
#
type cloudera_server_t;
type cloudera_server_exec_t;
init_daemon_domain(cloudera_server_t, cloudera_server_exec_t)
```

```
type cloudera_agent_t;
type cloudera_agent_exec_t;
init_daemon_domain(cloudera_agent_t, cloudera_agent_exec_t)

type cloudera_opt_t;
files_type(cloudera_opt_t)

type cloudera_lib_t;
files_type(cloudera_lib_t)

type hadoop_data_t;
files_type(hadoop_data_t)

type cloudera_server_var_run_t;
files_pid_file(cloudera_server_var_run_t)

type cloudera_agent_var_run_t;
files_pid_file(cloudera_agent_var_run_t)

type cloudera_server_tmp_t;
files_tmp_file(cloudera_server_tmp_t)

type cloudera_agent_tmp_t;
files_tmp_file(cloudera_agent_tmp_t)

type cloudera_server_var_lib_t;
files_var_lib_file(cloudera_server_var_lib_t)

type cloudera_agent_var_lib_t;
files_var_lib_file(cloudera_agent_var_lib_t)

########################################
#
# Declarations
#
type kinit_t;
domain_type(kinit_t)

type kinit_exec_t;
files_type(kinit_exec_t)

domain_entry_file(kinit_t, kinit_exec_t)
role system_r types kinit_t;

type kinit_tmp_t;
files_tmp_file(kinit_tmp_t)

########################################
#
# cloudera_agent local policy
#
allow cloudera_server_t self:capability { setuid sys_resource audit_write setgid
      dac_override };
allow cloudera_server_t self:process { execmem signal setrlimit setsched };
allow cloudera_server_t self:tcp_socket { setopt bind create getattr accept ioctl
      connect shutdown getopt listen };
allow cloudera_server_t self:udp_socket { ioctl create connect };
allow cloudera_server_t self:unix_dgram_socket { create connect };
```

```
files_tmp_filetrans(cloudera_server_t, cloudera_server_tmp_t, { file dir } )
manage_dirs_pattern(cloudera_server_t, cloudera_server_tmp_t, cloudera_server_tmp_t)
manage_files_pattern(cloudera_server_t, cloudera_server_tmp_t, cloudera_server_tmp_t)

files_var_lib_filetrans(cloudera_server_t, cloudera_server_var_lib_t, { file dir } )
manage_dirs_pattern(cloudera_server_t, cloudera_server_var_lib_t,
      cloudera_server_var_lib_t)
manage_files_pattern(cloudera_server_t, cloudera_server_var_lib_t,
      cloudera_server_var_lib_t)

files_pid_filetrans(cloudera_server_t, cloudera_server_var_run_t, { file dir })
manage_dirs_pattern(cloudera_server_t, cloudera_server_var_run_t,
      cloudera_server_var_run_t)
manage_files_pattern(cloudera_server_t, cloudera_server_var_run_t,
      cloudera_server_var_run_t)


list_dirs_pattern(cloudera_server_t, cloudera_opt_t, cloudera_opt_t)
read_files_pattern(cloudera_server_t, cloudera_opt_t, cloudera_opt_t)

# TODO one of these 2 interfaces should be uncommented for all of the
# allow cloudera_t domain:dir search;
# allow cloudera_t domain:file { read getattr open };
# denials. I think the dontaudit should be used
#domain_read_all_domains_state(cloudera_server_t)
#domain_dontaudit_read_all_domains_state(cloudera_server_t)

corenet_tcp_bind_generic_node(cloudera_server_t)
corenet_tcp_bind_generic_port(cloudera_server_t)

corecmd_exec_bin(cloudera_server_t)
corecmd_exec_shell(cloudera_server_t)
dev_read_rand(cloudera_server_t)
dev_read_urand(cloudera_server_t)
kernel_read_network_state(cloudera_server_t)
kernel_read_system_state(cloudera_server_t)

java_exec(cloudera_server_t)
logging_send_syslog_msg(cloudera_server_t)
sysnet_read_config(cloudera_server_t)
sssd_stream_connect(cloudera_server_t)

########################################
#
# cloudera_agent local policy
#
allow cloudera_agent_t self:capability { net_admin sys_ptrace dac_override };
allow cloudera_agent_t self:process { execmem signal signull sigkill setfscreate };
allow cloudera_agent_t self:netlink_route_socket { bind create getattr nlmsg_read };
allow cloudera_agent_t self:tcp_socket { setopt create getattr accept ioctl connect
      shutdown getopt };
allow cloudera_agent_t self:udp_socket { ioctl create connect };

files_tmp_filetrans(cloudera_agent_t, cloudera_agent_tmp_t, { file dir } )
manage_dirs_pattern(cloudera_agent_t, cloudera_agent_tmp_t, cloudera_agent_tmp_t)
manage_files_pattern(cloudera_agent_t, cloudera_agent_tmp_t, cloudera_agent_tmp_t)

files_var_lib_filetrans(cloudera_agent_t, cloudera_agent_var_lib_t, { file dir } )
manage_dirs_pattern(cloudera_agent_t, cloudera_agent_var_lib_t,
      cloudera_agent_var_lib_t)
manage_files_pattern(cloudera_agent_t, cloudera_agent_var_lib_t,
      cloudera_agent_var_lib_t)

manage_dirs_pattern(cloudera_agent_t, hadoop_data_t, hadoop_data_t)
manage_files_pattern(cloudera_agent_t, hadoop_data_t, hadoop_data_t)
manage_lnk_files_pattern(cloudera_agent_t, hadoop_data_t, hadoop_data_t)

exec_files_pattern(cloudera_agent_t, cloudera_lib_t, cloudera_lib_t)

read_files_pattern(cloudera_agent_t, cloudera_opt_t, cloudera_opt_t)

read_files_pattern(cloudera_agent_t, cloudera_server_t, cloudera_server_t)

# one of these 2 interfaces should be uncommented for all of the
# allow cloudera_t domain:dir search;
# allow cloudera_t domain:file { read getattr open };
# denials. I think the dontaudit should be used
#domain_read_all_domains_state(cloudera_agent_t)
#domain_dontaudit_read_all_domains_state(cloudera_agent_t)

corenet_tcp_connect_generic_port(cloudera_agent_t)
corecmd_exec_bin(cloudera_agent_t)
corecmd_exec_shell(cloudera_agent_t)
dev_read_urand(cloudera_agent_t)
kernel_read_network_state(cloudera_agent_t)
kernel_read_system_state(cloudera_agent_t)

java_exec(cloudera_agent_t)
libs_exec_lib_files(cloudera_agent_t)
miscfiles_read_localization(cloudera_agent_t)
sysnet_read_config(cloudera_agent_t)

########################################
#
# kinit local policy
#
allow kinit_t self:process setfscreate;
allow kinit_t self:tcp_socket { ioctl getopt create connect setopt };
allow kinit_t self:udp_socket { ioctl create connect };
allow kinit_t self:netlink_route_socket { bind create getattr nlmsg_read };

files_tmp_filetrans(kinit_t, kinit_tmp_t, file)
manage_files_pattern(kinit_t, kinit_tmp_t, kinit_tmp_t)

corenet_tcp_connect_kerberos_port(kinit_t)
dev_read_urand(kinit_t)
miscfiles_read_localization(kinit_t)
seutil_read_file_contexts(kinit_t)
sssd_read_public_files(kinit_t)
sysnet_read_config(kinit_t)
term_use_generic_ptys(kinit_t)

# THIS IS ONLY FOR TESTING WITHOUT REBOOTING MACHINE
# DO NOT USE IN PRODUCTION
gen_require(`
      type inetd_child_t;
')
domtrans_pattern(inetd_child_t, cloudera_server_exec_t, cloudera_server_t)
domtrans_pattern(inetd_child_t, cloudera_agent_exec_t, cloudera_agent_t)
domtrans_pattern(inetd_child_t, kinit_exec_t, kinit_t)
```

SOME COMMON ERRORS WHEN WRITING THE POLICY ABOVE:

1. Sometimes, xinetd could be running on the system and can causing issues with the login policy handled by openssh. Traditionally, openssh will spawn ssh users with the unconfined_t domain. Since xinted usually handles the initial communication, the transition may not working correctly and the user shell could be labeled as inetd_child_t. This should also be corrected for eventual SELinux confined users (e.g., sysadm_t, staff_t, or if a custom cloudera_admin_t doesn't yet exist.
2. One of the changes that needs to be made to the existing SELinux policy for Cloudera is that corenet_tcp_bind_generic_port() should be removed and access to ports should be labeled more specifically.

3.8 Summary

Hadoop and big data are no longer buzz words in large enterprises. Whether for the correct reasons or not, enterprise data warehouses are moving to Hadoop and along with it come petabytes of data.

In this section, we have laid the groundwork for conducting future security assessments on the Hadoop ecosystem and securing it. This is to ensure that Big Data in Hadoop does not become a big problem or a big target. Vendors pitch their technologies as the magical silver bullet. However, there are many challenges when it comes to deploying security controls in your Hadoop environment.

This section also provided the Hadoop threat model which the reader can further expand and customize to their organizational environment. It also provides a target reference architecture around Hadoop security and covers the entire control stack.

Hadoop and big data represent a green field opportunity for security practitioners. It provides a chance to get ahead of the curve, test and deploy your tools, processes, patterns, and techniques before big data becomes a big problem.

The following are some key recommendations in helping mitigate the security risks and threats identified in the Big Data ecosystem.

1. Select products and vendors that have proven experience in similar-scale deployments. Request vendor references for large deployments (that is, similar in size to your organization) that have been running the security controls under consideration for your project for at least one year
2. Key pillars are: Accountability, balancing network centric, access-control centric, and data centric security is absolutely critical in achieving a good overall trustworthy security posture.
3. Data-centric security, such as label security or cell-level security for sensitive data is preferred. Label security and cell-level security are integrated into the data or into the application code rather than adding data security after the fact

4. Externalize data security when possible and use data redaction, data masking or tokenization at the time of ingestion, or use data services with granular controls to access Hadoop
5. Harness the log and audit sprawl with data management tools, such as OSS Apache Falcon, Cloudera Navigator or the Zettaset Orchestrator. This helps achieve data provenance in the long run

4 Components of Hadoop

This chapter covers

- Managing files in HDFS
- Analyzing components of the MapReduce framework
- Reading and writing input and output data

In the last chapter we looked at setting up and installing Hadoop. We covered what the different nodes do and how to configure them to work with each other. Now that you have Hadoop running, let's look at the Hadoop framework from a programmer's perspective. If the previous chapter is like teaching you how to connect your turntable, your mixer, your amplifier, and your speakers together, then this chapter is about the techniques of mixing music.

We first cover HDFS, where you'll store data that your Hadoop applications will process. Next we explain the MapReduce framework in more detail. In chapter 1 we've already seen a MapReduce program, but we discussed the logic only at the conceptual level. In this chapter we get to know the Java classes and methods, as well as the underlying processing steps. We also learn how to read and write using different data formats.

4.1 Working with files in HDFS

HDFS is a filesystem designed for large-scale distributed data processing under frameworks such as MapReduce. You can store a big data set of (say) 100 TB as a single file in HDFS, something that would overwhelm most other filesystems. We discussed in chapter 2 how to replicate the data for availability and distribute it over multiple machines to enable parallel processing. HDFS abstracts these details away and gives you the illusion that you're dealing with only a single file.

As HDFS isn't a native Unix filesystem, standard Unix file tools, such as `ls` and `cp` don't work on it,[4] and neither do standard file read/write operations, such as `fopen()` and `fread()`. On the other hand, Hadoop does provide a set of command-line utilities that work similarly to the Linux file commands. In the next section we'll discuss those Hadoop file shell commands, which are your primary interface with the HDFS system. Section 3.1.2 covers Hadoop Java libraries for handling HDFS files programmatically.

NOTE A typical Hadoop workflow creates data files (such as log files) elsewhere and copies them into HDFS using one of the command-line utilities discussed in the next section. Your MapReduce programs then process this data, but they usually don't read any HDFS files directly. Instead they rely on the MapReduce framework to read and parse the HDFS files into individual records (key/value pairs), which are the unit of data MapReduce programs do work on. You rarely will have to programmatically read or write HDFS files except for custom import and export of data.

4.1.1 Basic file commands

Hadoop file commands take the form of

```
hadoop fs -cmd <args>
```

where `cmd` is the specific file command and `<args>` is a variable number of arguments. The command `cmd` is usually named after the corresponding Unix equivalent. For example, the command for listing files is[5]

```
hadoop fs –ls
```

Let's look at the most common file management tasks in Hadoop, which include

- Adding files and directories
- Retrieving files
- Deleting files

Uri for specifying exact file and directory location

Hadoop file commands can interact with both the HDFS filesystem and the local filesystem. (And as we'll see in chapter X, it can also interact with Amazon S3 as a filesystem.) A URI pinpoints the location of a specific file or directory. The full URI format is scheme://authority/path. The scheme is similar to a protocol. It can be hdfs or file, to specify the HDFS filesystem or the local filesystem, respectively. For HDFS, authority is the NameNode host and path is the path of the file or directory

[4] There are several ongoing projects that try to make HDFS mountable as a Unix filesystem. The primary project is FUSE (filesystem in userspace) for HDFS, but there are also projects for NFS mounting of HDFS partitions: http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.9/bk_user-guide/content/user-guide-hdfs-nfs.html.
[5] Some older documentation shows file utilities in the form of `hadoop dfs –cmd <args>`. Both `dfs` and `fs` are equivalent, although `fs` is the preferred form now.

of interest. For example, for a standard pseudo-distributed configuration running HDFS on the local machine on port 9000, a URI to access the example.txt file under the directory user/hia2 will look like hdfs://localhost:9000/user/hia2/example.txt. You can use the Hadoop cat command to show the content of that file:

```
hadoop fs -cat hdfs://localhost:9000/user/hia2/example.txt
```

As we'll see shortly, most setups don't need to specify the scheme://authority part of the URI. When dealing with the local filesystem, you'll probably prefer your standard Unix commands rather than the Hadoop file commands. For copying files between the local filesystem and HDFS, Hadoop commands such as `put` and `get` use the local filesystem as source and destination, respectively, without you specifying the file:// scheme. For other commands, if you leave out the scheme://authority part of the URI, the default from the Hadoop configuration is used. For example, if you have changed the et/hadoop/conf/core-site.xml file to the pseudo-distributed configuration, your fs.default.name property in the file should be

```
<property>
<name> fs.defaultFS</name>
<value>hdfs://localhost:9000</value>
</property>
```

Under this configuration, shorten the URI hdfs://localhost:9000/user/hia2/example.txt to /user/hia2/example.txt. Furthermore, HDFS defaults to a current working directory of /user/$USER, where $USER is your login user name. If you're logged in as chuck, then shorten the URI hdfs://localhost:9000/user/hia2/example.txt to example.txt. The Hadoop cat command to show the content of the file is

```
hadoop fs -cat example.txt
```

ADDING FILES AND DIRECTORIES

Before you can run Hadoop programs on data stored in HDFS, you'll need to put the data into HDFS first. Let's assume you've already formatted and started a HDFS filesystem. (For learning purposes, we recommend a pseudo-distributed configuration as a playground.) Let's create a directory and put a file in it.

HDFS has a default working directory of /user/$USER, where $USER is your login user name. This directory isn't automatically created for you, though, so let's create it with the `mkdir` command. For the purpose of illustration, we use hia2. You should substitute your user name in the example commands.

```
hadoop fs –mkdir /user/hia2
```

Hadoop's `mkdir` command automatically creates parent directories if they don't already exist, similar to the Unix `mkdir` command with the `-p` option. So the preceding command will create the /user directory too. Let's check on the directories with the `ls` command.

```
hadoop fs -ls /
```

You'll see this response showing the /user directory at the root / directory.

```
Found 1 items
drwxr-xr-x  - hia2 supergroup          0 2014-11-14 10:23 /user
```

If you want to see all the subdirectories, in a way similar to Unix's `ls` with the `-r` option, you can use Hadoop's `lsr` command.

```
hadoop fs -lsr /
```

You'll see all the files and directories recursively.

```
drwxr-xr-x      - hia2 supergroup       0 2014-11-14 10:23 /user
drwxr-xr-x      - hia2 supergroup       0 2014-11-14 10:23 /user/hia2
```

Now that we have a working directory, we can put a file into it. Create some text file on your local filesystem called example.txt. The Hadoop command `put` is used to copy files from the local system into HDFS.

```
hadoop fs -put example.txt .
```

Note the period (`.`) as the last argument in the command above. It means that we're putting the file into the default working directory. The command above is equivalent to

```
hadoop fs -put example.txt /user/hia2
```

We can re-execute the recursive file-listing command to see that the new file is added to HDFS.

```
$ hadoop fs –ls -R /
➥/user/hia2/example.txt

drwxr-xr-x      - hia2 supergroup       0 2014-11-14 10:23 /user
drwxr-xr-x      - hia2 supergroup       0 2014-11-14 11:02 /user/hia2
-rw-r--r--      1 hia2 supergroup       264 2014-11-14 11:02
```

Note that the –lsr variation of the command has been deprecated in Hadoop 2. In practice we don't need to check on all files recursively, and we may restrict ourselves to what's in our own working directory. We would use the Hadoop ls command in its simplest form:

```
$ hadoop fs -ls
Found 1 items
-rw-r--r--  1 hia2 supergroup        264 2009-01-14 11:02
➥/user/hia2/example.txt
```

The output displays properties, such as permission, owner, group, file size, and last modification date, all of which are familiar Unix concepts. The column stating "1" reports the replication factor of the file. It should always be 1 for the pseudo-distributed configuration. For production clusters, the replication factor is typically 3 but can be any positive integer. Replication factor is not applicable to directories, so they will only show a dash (-) for that column.

After you've put data into HDFS, you can run Hadoop programs to process it. The output of the processing will be a new set of files in HDFS, and you'll want to read or retrieve the results.

RETRIEVING FILES

The Hadoop command `get` does the exact reverse of `put`. It copies files from HDFS to the local filesystem. Let's say we no longer have the example.txt file locally and we want to retrieve it from HDFS; we can run the command

```
hadoop fs -get example.txt .
```

to copy it into our local current working directory.

Another way to access the data is to display it. The Hadoop `cat` command allows us to do that.

```
hadoop fs -cat example.txt
```

We can use the Hadoop `file` command with Unix pipes to send its output for further processing by other Unix commands. For example, if the file is huge (as typical Hadoop files are) and you're interested in a quick check of its content, you can pipe the output of Hadoop's cat into a Unix head.

```
hadoop fs -cat example.txt | head
```

Hadoop natively supports a `tail` command for looking at the last kilobyte of a file.

```
hadoop fs -tail example.txt
```

After you finish working with files in HDFS, you may want to delete them to free up space.

DELETING FILES

You shouldn't be too surprised by now that the Hadoop command for removing files is `rm`.

```
hadoop fs –rm example.txt
```

The `rm` command can also be used to delete empty directories.

LOOKING UP HELP

A list of Hadoop file commands, together with the usage and description of each command, is given in the appendix. For the most part, the commands are modeled after their Unix equivalent. You can execute `hadoop fs` (with no parameters) to get a complete list of all commands available on your version of Hadoop. You can also use `help` to display the usage and a short description of each command. For example, to get a summary of `ls`, execute

```
hadoop fs –help ls
```

and you should see the following description:

```
-ls [-d] [-h] [-R] [<path> ...]:        List the contents that match the specified
       file pattern. If  path is not specified, the contents of
       /user/<currentUser> will be listed. Directory entries are of the form
                 permissions - userid groupid size_of_directory(in bytes)
       modification_date(yyyy-MM-dd HH:mm) directoryName
       and file entries are of the form
         permissions number_of_replicas userid groupid size_of_file(in bytes)
       modification_date(yyyy-MM-dd HH:mm) fileName
       -d  Directories are listed as plain files.
       -h  Formats the sizes of files in a human-readable fashion
       rather than a number of bytes.
       -R  Recursively list the contents of directories.
```

Although the command-line utilities are sufficient for most of your interaction with the HDFS filesystem, they're not exhaustive and there'll be situations where you may want deeper access into the HDFS API. Let's see how to do so in the next section.

4.1.2 Reading and writing to HDFS programmatically

To motivate an examination of the HDFS Java API, we'll develop a PutMerge program for merging files while putting them into HDFS. The command-line utilities don't support this operation; we'll use the API.

The motivation for this example came when we wanted to analyze Apache log files coming from many web servers. We can copy each log file into HDFS, but in general, Hadoop works more effectively with a single large file rather than a number of smaller ones. ("Smaller" is relative here as it can still be tens or hundreds of gigabytes.) Besides, for analytics purposes we think of the log data as one big file. That it's spread over multiple files is an incidental result of the physical web server architecture. One solution is to merge all the files first and then copy the combined file into HDFS. Unfortunately, the file merging will require a lot of disk space in the local machine. It would be much easier if we could merge all the files on the fly as we copy them into HDFS.

What we need is, therefore, a PutMerge-type of operation. Hadoop's command-line utilities include a `getmerge` command for merging a number of HDFS files before copying them onto the local machine. What we're looking for is the exact opposite. This is not available in Hadoop's file utilities. We'll write our own program using the HDFS API.

The main classes for file manipulation in Hadoop are in the package org.apache.hadoop.fs. Basic Hadoop file operations include the familiar `open`, `read`, `write`, and `close`. In fact, the Hadoop file API is generic and can be used for working with filesystems other than HDFS. For our PutMerge program, we'll use the Hadoop file API to both read the local filesystem and write to HDFS.

The starting point for the Hadoop file API is the `FileSystem` class. This is an abstract class for interfacing with the filesystem, and there are different concrete subclasses for handling HDFS and the local filesystem. You get the desired `FileSystem` instance by calling the factory method `FileSystem.get(Configuration conf)`. The `Configuration` class is a special class for holding key/value configuration parameters. Its default instantiation is based on the resource configuration for your HDFS system. We can get the FileSystem object to interface with HDFS by

```
Configuration conf = new Configuration(); FileSystem hdfs = FileSystem.get(conf);
```

To get a `FileSystem` object specifically for the local filesystem, there's the `FileSystem.getLocal(Configuration conf)` factory method.

```
FileSystem local = FileSystem.getLocal(conf);
```

Hadoop file API uses Path objects to encode file and directory names and `FileStatus` objects to store metadata for files and directories. Our PutMerge program will merge all files from a local directory. We use the `FileSystem`'s `listStatus()` method to get a list of files in a directory.

```
Path inputDir = new Path(args[0]);
FileStatus[] inputFiles = local.listStatus(inputDir);
```

The length of the `inputFiles` array is the number of files in the specified directory. Each `FileStatus` object in `inputFiles` has metadata information such as file length, permissions, modification time, and others. Of interest to our PutMerge program is each file's `Path` representation, `inputFiles[i].getPath()`. We can use this `Path` to request an `FSDataInputStream` object for reading in the file.

```
FSDataInputStream in = local.open(inputFiles[i].getPath());
byte buffer[] = new byte[256];
int bytesRead = 0;
while( (bytesRead = in.read(buffer)) > 0) {
...
}
in.close();
```

`FSDataInputStream` is a subclass of Java's standard `java.io.DataInputStream` with additional support for random access. For writing to a HDFS file, there's the analogous `FSDataOutputStream` object.

```
Path hdfsFile = new Path(args[1]);
FSDataOutputStream out = hdfs.create(hdfsFile); out.write(buffer, 0, bytesRead);
out.close();
```

To complete the PutMerge program, we create a loop that goes through all the files in `inputFiles` as we read each one in and write it out to the destination HDFS file. You can see the complete program in Listing 4.1.

Listing 4.1 A putMerge program

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration; import
      org.apache.hadoop.fs.FSDataInputStream; import
      org.apache.hadoop.fs.FSDataOutputStream; import
      org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class PutMerge {

    public static void main(String[] args) throws IOException {

        Configuration conf = new Configuration();
        FileSystem hdfs = FileSystem.get(conf);
        FileSystem local = FileSystem.getLocal(conf);

        Path inputDir = new Path(args[0]);                   1
        Path hdfsFile = new Path(args[1]);

        try {
            FileStatus[] inputFiles = local.listStatus(inputDir);       2
            FSDataOutputStream out = hdfs.create(hdfsFile);             3

            for (int i=0; i<inputFiles.length; i++) {
                System.out.println(inputFiles[i].getPath().getName());
                FSDataInputStream in =
➥ local.open(inputFiles[i].getPath());                      4
                byte buffer[] = new byte[256];
                int bytesRead = 0;
                while( (bytesRead = in.read(buffer)) > 0) {
                    out.write(buffer, 0, bytesRead);
                }
                in.close();
            }
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1 Specify input directory and output file
2 Get list of local files
3 Create HDFS output stream
4 Open local input stream

The general flow of the program involves first setting the local directory and the HDFS destination file based on user-specified arguments #1. In #2 we extract information about each file in the local input directory. We create an output stream to write to the HDFS file in #3. We loop through each file in the local directory, and #4 opens an input stream to read that file. The rest of the code is standard Java file copy.

The FileSystem class also has methods such as `delete()`, `exists()`, `mkdirs()`, and `rename()` for other standard file operations. You can find the most recent Javadoc for the Hadoop file API at

http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/fs/package-summary.html .

We have covered how to work with files in HDFS. You now know a few ways to put data into and out of HDFS. But merely having data isn't terribly interesting. You want to process it, analyze it, and do other things. Let's conclude our discussion of HDFS and move on to the other major component of Hadoop, the MapReduce framework, and how to program under it.

4.2 Anatomy of a MapReduce program

As we have mentioned before, a MapReduce program processes data by manipulating (key/value) pairs in the general form

map: (K1,V1) → list(K2,V2)

reduce: (K2,list(V2)) → list(K3,V3)

Not surprisingly, this is an overly generic representation of the data flow. In this section we learn more details about each stage in a typical MapReduce program. Figure 4.1 displays a high-level diagram of the entire process, and we further dissect each component as we step through the flow.

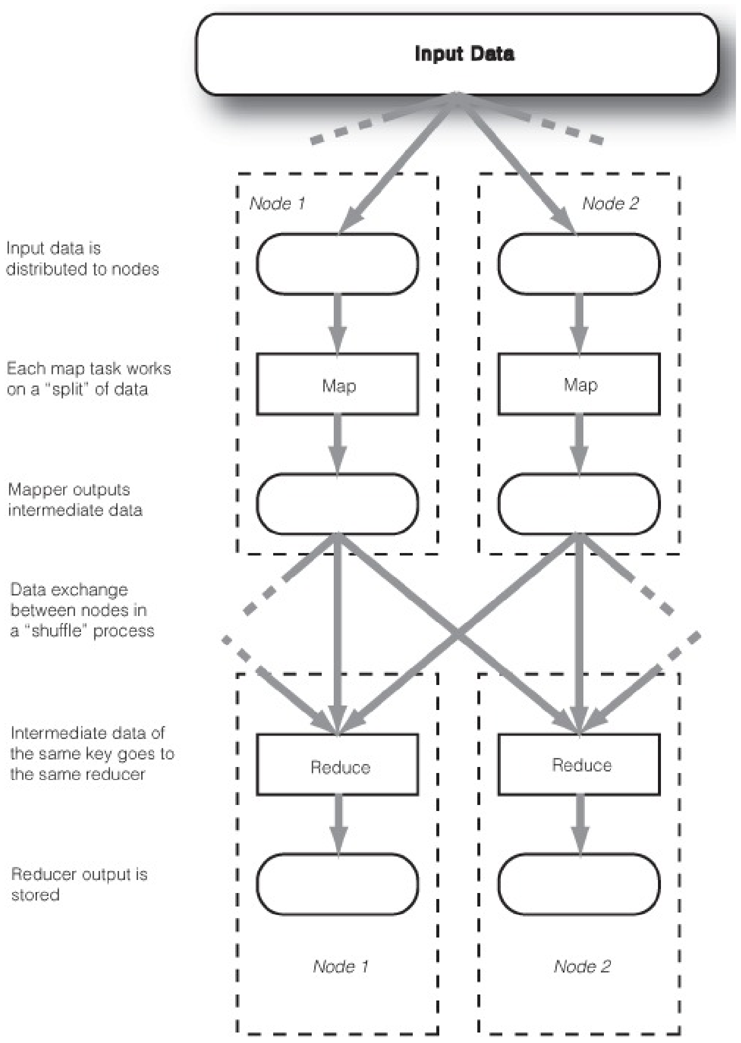

Input Data
Node 1
Node 2
Input data is distributed to nodes
Each map task works on a "split" of data
Map
Map
Mapper outputs intermediate data
Data exchange between nodes in a "shuffle" process
Intermediate data of the same key goes to the same reducer
Reduce
Reduce
Reducer output is stored
Node 1
Node 2

Figure 4.1 T he general MapReduce data flow. Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the "shuffle" step. This restriction on communication greatly helps scalability.

Before we analyze how data gets passed onto each individual stage, we should first familiarize ourselves with the data types that Hadoop supports.

4.2.1 Hadoop data types

Despite our many discussions regarding keys and values, we have yet to mention their types. The MapReduce framework won't allow them to be any arbitrary class. For example, although we can and often do talk about certain keys and values as integers, strings, and so on, they aren’t exactly standard Java classes, such as `Integer`, `String`, and so forth. This is because the MapReduce framework has a certain defined way of serializing the key/value pairs to move them across the cluster’s network, and only classes that support this kind of serialization can function as keys or values in the framework.

More specifically, classes that implement the Writable interface can be values, and classes that implement the `WritableComparable<T>` interface can be either keys or values. Note that the `WritableComparable<T>` interface is a combination of the `Writable` and `java.lang.Comparable<T>` interfaces. We need the comparability requirement for keys because they will be sorted at the reduce stage, whereas values are simply passed through.

Hadoop comes with a number of predefined classes that implement `WritableComparable`, including wrapper classes for all the basic data types, as seen in Table 4.1.

Table 4.1 List of frequently used types for the key/value pairs. These classes all implement the `WritableComparable` interface.

Class	Description
`BooleanWritable`	Wrapper for a standard Boolean variable
`ByteWritable`	Wrapper for a single byte
`BytesWritable`	Wrapper for a byte array
`DoubleWritable`	Wrapper for a Double
`FloatWritable`	Wrapper for a Float
`IntWritable`	Wrapper for a Integer
`LongWritable`	Wrapper for a Long
`Text`	Wrapper to store text using the UTF8 format
`NullWritable`	Placeholder when the key or value is not needed

Keys and values can take on types beyond the basic ones that Hadoop natively supports. You can create your own custom type as long as it implements the Writable (or `WritableComparable<T>`) interface. For example, Listing 4.2 shows a class that can represent edges in a network. This may represent a flight route between two cities.

Listing 4.2 An example class that implements the `WritableComparable` interface

```
public class Edge implements WritableComparable<Edge>{

    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode() { return departureNode;}

    @Override
    public void readFields(DataInput in) throws IOException {      1
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOException {      2
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {                      3
        return (departureNode.compareTo(o.departureNode) != 0)
            ? departureNode.compareTo(o.departureNode)
            : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

1 Specify how to read data in
2 Specify how to write data out
3 Define ordering of data

The Edge class implements the `readFields()` #1 and `write()` #2 methods of the Writable interface. They work with the Java `DataInput` and `DataOutput` classes to serialize the class contents. Implement the `compareTo()` method #3 for the `Comparable` interface. It returns -1, 0, or 1 if the called `Edge` is less than, equal to, or greater than the given `Edge`.

With the data type interfaces now defined, we can proceed to the first stage of the data flow process as described in Figure 4.1: the mapper.

4.2.2 Mapper

To serve as the mapper, a class extends the `Mapper` class. The `Mapper` class is responsible for the data-processing step. It utilizes Java generics of the form `Mapper<K1,V1,K2,V2>` where the key classes and value classes implement the `WritableComparable` and `Writable` interfaces, respectively. Its single method is to process an individual (key/value) pair:

```
void map(K1 key,
         V1 value,
         Context context
         ) throws IOException
```

The function generates a (possibly empty) list of `(K2, V2) pairs` for a given `(K1, V1)` input pair. The `Context` receives the output of the mapping process as well as providing the option to record extra information about the mapper as the task progresses.

Hadoop provides a few useful mapper implementations. You can see some of them in the Table 4.2.

Table 4.2 Some useful `Mapper` implementations predefined by Hadoop

Class	Description
`InverseMapper<K,V>`	Implements `Mapper<K,V,V,K>` and reverses the key/value pair
`RegexMapper<K>`	Implements `Mapper<K,Text,Text,LongWritable>` and generates a (match, 1) pair for every regular expression match
`TokenCounterMapper<K>`	Implements `Mapper<K,Text,Text,LongWritable>` and generates a (token, 1) pair when the input value is tokenized

As the MapReduce name implies, the major data flow operation after map is the reduce phase, shown in the bottom part of Figure 4.1.

4.2.3 Reducer

As with any mapper implementation, a reducer must first extend the MapReduce base class to allow for configuration and cleanup. In addition, it must also implement the Reducer interface which has the following single method:

```
void reduce(K2 key,
            Iterator<V2> values,
            Context context
            ) throws IOException
```

When the reducer task receives the output from the various mappers, it sorts the incoming data on the key of the (key/value) pair and groups together all values of the same key. The `reduce()` function is then called, and it generates a (possibly empty) list of `(K3, V3)` pairs by iterating over the values associated with a given key. The `Context` receives the output of the reduce process and writes it to an output file. The `Context` also provides the option to record extra information about the reducer as the task progresses.

Table 4.3 lists a couple of basic reducer implementations provided by Hadoop.

Table 4.3 Some useful Reducer implementations predefined by Hadoop

Class	Description
`IntSumReducer<K>`	Implements `Reducer<K,IntWritable,K,IntWritable>` and determines the sum of all values corresponding to the given key
`LongSumReducer<K>`	Implements `Reducer<K,LongWritable,K,LongWritable>` and determines the sum of all values corresponding to the given key

Although we have referred to Hadoop programs as MapReduce applications, there is a vital step between the two stages: directing the result of the mappers to the different reducers. This is the responsibility of the partitioner.

4.2.4 Partitioner—redirecting output from Mapper

A common misconception for first-time MapReduce programmers is to use only a single reducer. After all, a single reducer sorts all of your data before processing— and who doesn't like sorted data? Our discussions regarding MapReduce expose the folly of such thinking. We would have ignored the benefits of parallel computation. With one reducer, our compute cloud has been demoted to a compute raindrop.

With multiple reducers, we need some way to determine the appropriate one to send a (key/value) pair outputted by a mapper. The default behavior is to hash the key to determine the reducer. Hadoop enforces this strategy by use of the `HashPartitioner` class. Sometimes the `HashPartitioner` will steer you awry. Let's return to the `Edge` class introduced in section 3.2.1.

Suppose you used the `Edge` class to analyze flight information data to determine the number of passengers departing from each airport. Such data may be

(San Francisco, Los Angeles)   Chuck Lam

(San Francisco, Dallas) Mark Davis

...

If you used `HashPartitioner`, the two rows could be sent to different reducers. The number of departures would be processed twice and both times erroneously.

How do we customize the partitioner for your applications? In this situation, we want all edges with a common departure point to be sent to the same reducer. This is done easily enough by hashing the `departureNode` member of the `Edge`:

```
public class EdgePartitioner implements Partitioner<Edge, Writable>
{
    @Override
    public int getPartition(Edge key, Writable value, int numPartitions)
    {
        return key.getDepartureNode().hashCode() % numPartitions;
    }
}
```

A custom partitioner only needs to implement one function: `getPartition()`. The function returns an integer between 0 and the number of reduce tasks indexing to which reducer the (key/value) pair will be sent. If the configuration state is needed (for instance, to get configuration parameters), the class must implement the `Configurable` interface.

The exact mechanics of the partitioner may be difficult to follow. Figure 4.2 illustrates this for better understanding.

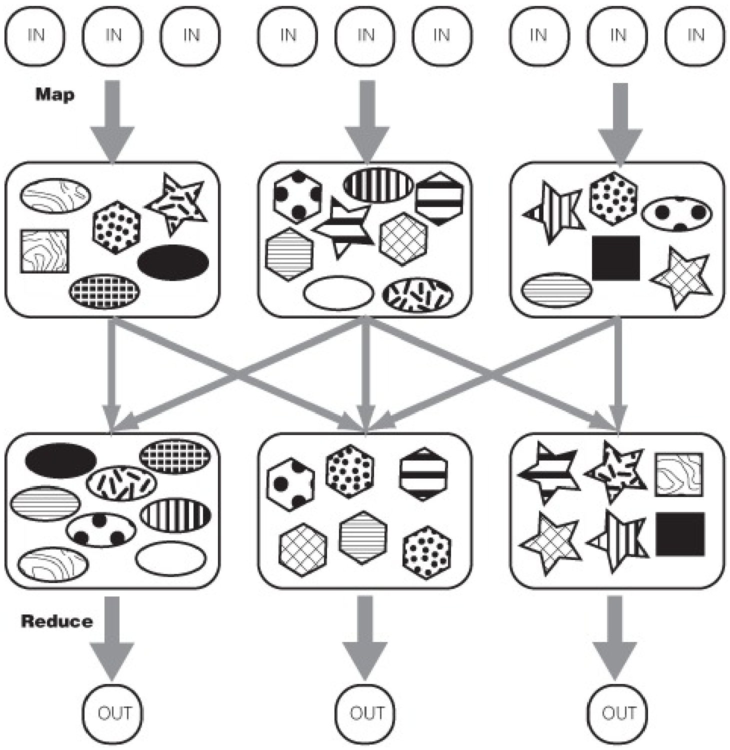


Figure 4.2 The MapReduce data flow, with an emphasis on partitioning and shuffling. Each icon is a key/value pair. The shapes represent keys, whereas the inner patterns represent values. After shuffling, all icons of the same shape (key) are in the same reducer. Different keys can go to the same reducer, as seen in the rightmost reducer. The partitioner decides which key goes where. Note that the leftmost reducer is more loaded due to more data under the “ellipse” key.

Between the map and reduce stages, a MapReduce application must take the output from the mapper tasks and distribute the results among the reducer tasks. This process is typically called *shuffling*, because the output of a mapper on a single node may be sent to reducers across multiple nodes in the cluster.

4.2.5 Combiner—local reduce

In many situations with MapReduce applications, we may wish to perform a “local reduce” before we distribute the mapper results. Consider the `WordCount` example of chapter 1 once more. If the job processes a document containing the word “the” 574 times, it’s much more efficient to store and shuffle the pair (“the”, 574) once instead of the pair (“the”, 1) multiple times. This processing step is known as *combining*. We explain combiners in more depth in section 4.6.

4.2.6 *Word counting with predefined mapper and reducer classes*

We have concluded our preliminary coverage of all the basic components of MapReduce. Now that you've seen more classes provided by Hadoop, it'll be fun to revisit the WordCount example (see Listing 4.3), using some of the classes we've learned.

Listing 4.3 Revised version of the WordCount example

```
public class WordCount2 extends Configured implements Tool {

     @Override
     public int run(String[] args) throws Exception {

              Configuration conf = getConf();
              Job job = Job.getInstance(conf, "wc");

              FileInputFormat.addInputPath(job, new Path(args[0]));
              FileOutputFormat.setOutputPath(job, new Path(args[1]));

              job.setOutputKeyClass(Text.class);
              job.setOutputValueClass(LongWritable.class);
              job.setMapperClass(TokenCounterMapper.class);     1
              job.setCombinerClass(LongSumReducer.class);               2
              job.setReducerClass(LongSumReducer.class);

              System.exit(job.waitForCompletion(true) ? 0 : 1);

              return 0;
     }

     public static void main(String[] args) throws Exception {
     int res = ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(res);
     }
}
```

1 Hadoop's own TokenCounterMapper
2 Hadoop's own LongSumReducer

We have to write only the driver for this MapReduce program because we have used Hadoop's predefined `TokenCountMapper` class #1 and `LongSumReducer` class #2. Easy, isn't it? Hadoop provides the ability to generate more sophisticated programs (this will be the focus of part 2 of the book), but we want to emphasize that Hadoop allows you to rapidly generate useful programs with a minimal amount of code.

4.3 *Reading and writing*

Let's see how MapReduce reads input data and writes output data and focus on the file formats it uses. To enable easy distributed processing, MapReduce makes certain assumptions about the data it's processing. It also provides flexibility in dealing with a variety of data formats.

Input data usually resides in large files, typically tens or hundreds of gigabytes or even more. One of the fundamental principles of MapReduce's processing power is the splitting of the input data into chunks. You can process these chunks in parallel using multiple machines. In Hadoop terminology these chunks are called input splits.

The size of each split should be small enough for a more granular parallelization. (If all the input data is in one split, then there is no parallelization.) On the other hand, each split shouldn't be so small that the overhead of starting and stopping the processing of a split becomes a large fraction of execution time.

The principle of dividing input data (which often can be one single massive file) into splits for parallel processing explains some of the design decisions behind Hadoop's generic FileSystem as well as HDFS in particular. For example, Hadoop's FileSystem provides the class `FSDataInputStream` for file reading rather than using Java's `java.io.DataInputStream`. `FSDataInputStream` extends `DataInputStream` with random read access, a feature that MapReduce requires because a machine may be assigned to process a split that sits right in the middle of an input file. Without random access, it would be extremely inefficient to have to read the file from the beginning until you reach the location of the split. You can also see how HDFS is designed for storing data that MapReduce will split and process in parallel. HDFS stores files in blocks spread over multiple machines. Roughly speaking, each file block is a split. As different machines will likely have different blocks, parallelization is automatic if each split/block is processed by the machine that it's residing at. Furthermore, as HDFS replicates blocks in multiple nodes for reliability, MapReduce can choose any of the nodes that have a copy of a split/block.

Input splits and record boundaries

Note that input splits are a logical division of your records whereas HDFS blocks are a physical division of the input data. It's extremely efficient when they're the same but in practice it's never perfectly aligned. Records may cross block boundaries. Hadoop guarantees the processing of all records. A machine processing a particular split may fetch a fragment of a record from a block other than its "main" block and which may reside remotely. The communication cost for fetching a record fragment is inconsequential because it happens relatively rarely.

You'll recall that MapReduce works on key/value pairs. So far we've seen that Hadoop by default considers each line in the input file to be a record and the key/value pair is the byte offset (key) and content of the line (value), respectively. You may not have recorded all your data that way. Hadoop supports a few other data formats and allows you to define your own.

4.3.1 InputFormat

The way an input file is split up and read by Hadoop is defined by one of the implementations of the `InputFormat` interface. `TextInputFormat` is the default `InputFormat` implementation, and it's the data format we've been implicitly using up to now. It's often useful for input data that has no definite key value, when you want to get the content one line at a time. The key returned by TextInputFormat is the byte offset of each line, and we have yet to see any program that uses that key for its data processing.

POPULAR INPUTFORMAT CLASSES

Table 4.4 lists other popular implementations of InputFormat along with a description of the key/value pair each one passes to the mapper.

Table 4.4 Main InputFormat classes. TextInputFormat is the default unless an alternative is specified. The object type for key and value are also described.

InputFormat	Description
TextInputFormat	Each line in the text files is a record. Key is the byte offset of the line, and value is the content of the line. key: LongWritable value: Text
KeyValueTextInputFormat	Each line in the text files is a record. The first separator character divides each line. Everything before the separator is the key, and everything after is the value. The separator is set by the key.value.separator.in.input.line property, and the default is the tab (\t) character. key: Text value: Text
SequenceFileInputFormat<K,V>	An InputFormat for reading in sequence files. Key and value are user defined. Sequence file is a Hadoop-specific compressed binary file format. It's optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. key: K (user defined) value: V (user defined)
NLineInputFormat	Same as TextInputFormat, but each split is guaranteed to have exactly *N* lines. The mapred.line.input.format.linespermap property, which defaults to one, sets *N*. key: LongWritable value: Text

KeyValueTextInputFormat is used in the more structured input files where a predefined character, usually a tab (\t), separates the key and value of each line (record). For example, you may have a tab-separated data file of timestamps and URLs:

```
17:16:18   http://hadoop.apache.org/core/docs/r0.19.0/api/index.html
17:16:19   http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html
17:16:20   http://wiki.apache.org/hadoop/GettingStartedWithHadoop
17:16:20   http://www.maxim.com/hotties/2008/finalist_gallery.aspx
```

```
17:16:25    http://wiki.apache.org/hadoop/
...
```

You can set your `Job` object to use the `KeyValueTextInputFormat` class to read this file.

```
job.setInputFormatClass(KeyValueTextInputFormat.class);
```

Given the preceding example file, the first record your mapper reads will have a key of "17:16:18" and a value of "http://hadoop.apache.org/core/docs/r0.19.0/api/ index.html". The second record to your mapper will have a key of "17:16:19" and a value of "http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html." And so on.

Recall that our previous mappers had used `LongWritable` and `Text` as the key and value types, respectively. `LongWritable` is a reasonable type for the key under `TextInputFormat` because the key is a numerical offset. When using `KeyValueTextInputFormat`, both the key and the value will be of type `Text`, and you'll have to change your Mapper implementation and `map()` method to reflect the new key type.

The input data to your MapReduce job does not necessarily have to be some external data. In fact it's often the case that the input to one MapReduce job is the output of some other MapReduce job. As we'll see, you can customize your output format too. The default output format writes the output in the same format that `KeyValueTextInputFormat` can read back in (i.e., each line is a record with key and value separated by a tab character). Hadoop provides a much more efficient binary compressed file format called sequence file. This sequence file is optimized for Hadoop processing and should be the preferred format when chaining multiple MapReduce jobs. The `InputFormat` class to read sequence files is `SequenceFileInputFormat`. The object type for key and value in a sequence file are definable by the user. The output and the input type have to match, and your Mapper implementation and `map()` method have to take in the right input type.

CREATING A CUSTOM INPUTFORMAT—INPUTSPLIT AND RECORDREADER

Sometimes you may want to read input data in a way different from the standard `InputFormat` classes. In that case you'll have to write your own custom `InputFormat` class. Let's look at what it involves. `InputFormat` is an interface consisting of only two methods.

```
public interface InputFormat<K, V> {
  List<InputSplit> getSplits(JobContext context)
               throws IOException, InterruptedException;
  RecordReader<K, V> createRecordReader(InputSplit split,
               TaskAttemptContext context)
               throws IOException, InterruptedException;
}
```

The two methods sum up the functions that `InputFormat` has to perform:

- Identify all the files used as input data and divide them into input splits. Each map task is assigned one split.
- Provide an object (`RecordReader`) to iterate through records in a given split, and to parse each record into key and value of predefined types.

Who wants to worry about how files are divided into splits? In creating your own InputFormat class you should subclass the FileInputFormat class, which takes care of file splitting. In fact, all the InputFormat classes in Table 4.4 subclass FileInputFormat. FileInputFormat implements the getSplits() method but leaves createRecordReader() abstract for the subclass to fill out. FileInputFormat's getSplits() implementation is subject to the constraints that each split must have more than mapreduce.input.fileinputformat.split.maxsize number of bytes but also be smaller than the block size of the filesystem. In practice, a split usually ends up being the size of a block, which defaults to 64 MB in HDFS.

FileInputFormat has a number of protected methods a subclass can overwrite to change its behavior, one of which is the isSplitable(FileSystem fs, Path filename) method. It checks whether you can split a given file. The default implementation always returns true, so all files larger than a block will be split. Sometimes you may want a file to be its own split, and you'll overwrite isSplitable() to return false in those situations. For example, some file compression schemes don't support splits. (You can't start reading from the middle of those files.) Some data-processing operations, such as file conversion, will need to treat each file as an atomic record and one should also not be able to split it.

In using FileInputFormat you focus on customizing RecordReader, which is responsible for parsing an input split into records and then parsing each record into a key/value pair. Let's look at the signature of this interface.

```
public interface RecordReader<K, V> {
     boolean nextKeyValue(K key, V value) throws IOException, InterruptedException;

  K createKey();
  V createValue();

     float getProgress()throws IOException, InterruptedException;
     public void close() throws IOException;
}
```

Instead of writing our own RecordReader, we'll again leverage existing classes provided by Hadoop. For example, LineRecordReader implements RecordReader <LongWritable,Text>. It's used in TextInputFormat and reads one line at a time, with byte offset as key and line content as value. KeyValueLineRecordReader uses KeyValueTextInputFormat. For the most part, your custom RecordReader will be a wrapper around an existing implementation, and most of the action will be in the next() method.

One use case for writing your own custom InputFormat class is to read records in a specific type rather than the generic Text type. For example, we had previously used KeyValueTextInputFormat to read a tab-separated data file of timestamps and URLs. The class ends up treating both the timestamp and the URL as Text type. For our illustration, let's create a TimeUrlTextInputFormat that works exactly the same but treats the URL as a URLWritable type.[6] As mentioned earlier, we create our InputFormat class by extending FileInputFormat and implementing the factory method to return our RecordReader.

```
public class TimeUrlTextInputFormat extends
➥    FileInputFormat<Text, URLWritable> {

  public RecordReader<Text, URLWritable> getRecordReader(
➥    InputSplit input, JobConf job, Reporter reporter)
➥    throws IOException {

    return new TimeUrlLineRecordReader(job, (FileSplit)input);

  }
}
```

Our URLWritable class is quite straightforward:

```
public class URLWritable implements Writable {

  protected URL url;

  public URLWritable() { }

  public URLWritable(URL url) {
    this.url = url;
  }

  public void write(DataOutput out) throws IOException {
    out.writeUTF(url.toString());
  }

  public void readFields(DataInput in) throws IOException {
    url = new URL(in.readUTF());
  }

  public void set(String s) throws MalformedURLException {
    url = new URL(s);
  }
}
```

Our TimeUrlLineRecordReader will implement the six methods in the RecordReader interface, in addition to the class constructor. It's mostly a wrapper around KeyValueTextInputFormat, but converts the record value from Text to type URLWritable.

```
class TimeUrlLineRecordReader implements RecordReader<Text, URLWritable> {

  private KeyValueLineRecordReader lineReader;
  private Text lineKey, lineValue;

  public TimeUrlLineRecordReader(JobConf job, FileSplit split) throws
```

[6] We may also want the time key to be some type other than Text. For example, we can make up a type CalendarWritableComparable for it. We leave that as an exercise for the reader as we focus on a simpler illustration.

```
➥IOException {
    lineReader = new KeyValueLineRecordReader(job, split);

    lineKey = lineReader.createKey();
    lineValue = lineReader.createValue();
  }

  public boolean next(Text key, URLWritable value) throws IOException {
    if (!lineReader.next(lineKey, lineValue)) {
      return false;
    }

    key.set(lineKey);
    value.set(lineValue.toString());

    return true;
  }

  public Text createKey() {
    return new Text("");
  }

  public URLWritable createValue() {
    return new URLWritable();
  }

  public long getPos() throws IOException {
    return lineReader.getPos();
  }

  public float getProgress() throws IOException {
    return lineReader.getProgress();
  }

  public void close() throws IOException {
    lineReader.close();
  }
}
```

Our `TimeUrlLineRecordReader` class creates a `KeyValueLineRecordReader` object and passes the `getPos()`, `getProgress()`, and `close()` method calls directly to it. The `next()` method casts the `lineValue` `Text` object into the `URLWritable` type.

4.3.2 OutputFormat

MapReduce outputs data into files using the `OutputFormat` class, which is analogous to the `InputFormat` class. The output has no splits, as each reducer writes its output only to its own file. The output files reside in a common directory and are typically named part-*nnnnn*, where *nnnnn* is the partition ID of the reducer. `RecordWriter` objects format the output and `RecordReader`s parse the format of the input.

Hadoop provides several standard implementations of `OutputFormat`, as shown in Table 4.5. Not surprisingly, almost all the ones we deal with inherit from the `FileOutputFormat` abstract class; `InputFormat` classes inherit from `FileInputFormat`. You specify the `OutputFormat` by calling `setOutputFormat()` of the `JobConf` object that holds the configuration of your MapReduce job.

NOTE You may wonder why there's a separation between `OutputFormat` (`InputFormat`) and `FileOutputFormat` (`FileInputFormat`) when it seems all `OutputFormat` (`InputFormat`) classes extend `FileOutputFormat` (`FileInputFormat`). Are there `OutputFormat` (`InputFormat`) classes that don't work with files? Well, the `NullOutputFormat` implements `OutputFormat` in a trivial way and doesn't need to subclass `FileOutputFormat`. More importantly, there are `OutputFormat` (`InputFormat`) classes that work with databases rather than files, and these classes are in a separate branch in the class hierarchy from `FileOutputFormat` (`FileInputFormat`). These classes have specialized applications, and the interested reader can dig further in the online Java documentation for `DBInputFormat` and `DBOutputFormat`.

Table 4.5 Main `OutputFormat` classes. `TextOutputFormat` is the default.

OutputFormat	**Description**
`TextOutputFormat<K,V>`	Writes each record as a line of text. Keys and values are written as strings and separated by a tab (\t) character, which can be changed in the `mapred.textoutputformat.separator` property.
`SequenceFileOutputFormat<K,V>`	Writes the key/value pairs in Hadoop's proprietary sequence file format. Works in conjunction with `SequenceFileInputFormat`.
`NullOutputFormat<K,V>`	Outputs nothing.

The default `OutputFormat` is `TextOutputFormat`, which writes each record as a line of text. Each record's key and value are converted to strings through `toString()`, and a tab (\t) character separates them. The separator character can be changed in the `mapred.textoutputformat.separator` property.

`TextOutputFormat` outputs data in a format readable by `KeyValueTextInputFormat`. It can also output in a format readable by `TextInputFormat` if you make the key type a `NullWritable`. In that case the key in the key/value pair is not written out, and neither is the separator character. If you want to suppress the output completely, then you should use the `NullOutputFormat`. Suppressing the Hadoop output is useful if your reducer writes its output in its own way and doesn't need Hadoop to write any additional files.

Finally, `SequenceFileOutputFormat` writes the output in a sequence file format that can be read back in using `SequenceFileInputFormat`. It's useful for writing intermediate data results when chaining MapReduce jobs.

4.4 Summary

Hadoop is a software framework that demands a different perspective on data processing. It has its own filesystem, HDFS, that stores data in a way optimized for data-intensive processing. You need specialized Hadoop tools to work with HDFS, but fortunately most of those tools follow familiar Unix or Java syntax.

The data-processing part of the Hadoop framework is better known as MapReduce. Although the highlight of a MapReduce program is, not surprisingly, the Map and the Reduce operations, other operations done by the framework, such as data splitting and shuffling, are crucial to how the framework works. You can customize the other operations, such as Partitioning and Combining. Hadoop provides options for reading data and also to output data of different formats.

Now that we have a better understanding of how Hadoop works, let’s go on to part 2 of this book and look at various techniques for writing practical programs using Hadoop.

Part 2

Hadoop in Action

Part 2 teaches the practical skills required to write and run data-processing programs in Hadoop. We explore various examples of using Hadoop to analyze a patent data set, including advanced algorithms such as the Bloom filter. We also cover programming and administration techniques that are uniquely useful to working with Hadoop in production.

5 Writing basic MapReduce programs

This chapter covers

- Patent data as an example data set to process with Hadoop
- Skeleton of a MapReduce program
- Basic MapReduce programs to count statistics
- Hadoop's Streaming API for writing MapReduce programs using scripting languages
- Combiner to improve performance

The MapReduce programming model is unlike most programming models you may have learned. It'll take some time and practice to gain familiarity. To help develop your proficiency, we go through many example programs in the next couple chapters. These examples will illustrate various MapReduce programming techniques. By applying MapReduce in multiple ways you'll start to develop an intuition and a habit of "MapReduce thinking." The examples will cover simple tasks to advanced uses. In one of the advanced applications we introduce the Bloom filter, a data structure not normally taught in the standard computer science curriculum. You'll see that processing large data sets, whether you're using Hadoop or not, often requires a rethinking of the underlying algorithms.

We assume you already have a basic grasp of Hadoop. You can set up Hadoop, and you have compiled and run an example program, such as word counting from chapter 1. Let's use examples—from a real-world data set.

5.1 Getting the patent data set

To do anything meaningful with Hadoop we need data. Many of our examples will use patent data sets, both of which are available from the National Bureau of Economic Research (NBER) at http://www.nber.org/patents/. The data sets were originally compiled for the paper "The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools."[7] We use the citation data set acite75_99.zip and the patent description data set apat63_99.zip. acite75_99.zip unzips to cite75_99.txt and apat63_99.zip unzips to apat63_99.txt.

NOTE The data sets are approximately 250 MB each, which are small enough to make our examples runnable in Hadoop's standalone or pseudo-distributed mode. You can practice writing MapReduce programs using them even when you don't have access to a live cluster. The best part of Hadoop is that you can be fairly sure your MapReduce program will run on clusters of machines processing data sets 100 or 1,000 times larger with virtually no code changes.

A popular development tactic is to create a smaller, sampled subset of your large production data and call it the development data set. This development data set may only have several hundred megabytes. You develop your program in standalone or pseudo-distributed mode with the development data set. This gives your development process a fast turnaround time, the convenience of running on your own machine, and an isolated environment for debugging.

We have chosen these two data sets for our example programs because they're similar to most data types you'll encounter. First of all, the citation data encodes a graph in the same vein that web links and social networks are also graphs. Patents are published in chronological order; some of their properties resemble time series. Each patent is linked with a person (inventor) and a location (country of inventor). You can view them as personal or geographical data. Finally, you can look at the data as generic database relations with well-defined schemas, in a simple comma-separated format.[8]

5.1.1 The patent citation data

The patent citation data set contains citations from U.S. patents issued between 1975 and 1999. It has more than 16 million rows and the first few lines resemble the following:

```
"CITING","CITED"
3858241,956203
3858241,1324234
3858241,3398406
3858241,3557384
```

[7] NBER Working Paper 8498, by Hall, B. H., A. B. Jaffe, and M. Tratjenberg (2001).
[8] There are more common data types than two data sets can possibly represent. An important one that's missing here is text, but you've already seen text used in the word count example. Other missing types include XML, image, and geolocation (the lat-long variety). Math matrix is not represented in general, although the citation graph can be interpreted as a sparse 0/1 matrix.

```
3858241,3634889
3858242,1515701
3858242,3319261
3858242,3668705
3858242,3707004
...
```

The data set is in the standard comma-separated values (CSV) format, with the first line a description of the columns. Each of the other lines record one particular citation. For example, the second line shows that patent 3858241 cites patent 956203. The file is sorted by the citing patent. We can see that patent 3858241 cites five patents in total. Analyzing the data more quantitatively will give us deeper insights into it.

If you're only reading the data file, the citation data appears to be a bunch of numbers. You can "think" of this data in more interesting terms. One way is to visualize it as a graph. In figure 4.1 we've shown a portion of this citation graph. We can see that some patents are cited often whereas others aren't cited at all.[9] Patents like 5936972 and 6009552 cite a similar set of patents (4354269, 4486882, 5598422) even though they don't cite each other. We use Hadoop to derive descriptive statistics about this patent data, as well as look for interesting patterns that aren't immediately obvious.

[9] As with any data analysis, we must be careful when interpreting with limited data. When a patent doesn't seem to cite any other patents, it may be an older patent for which we have no citation information.

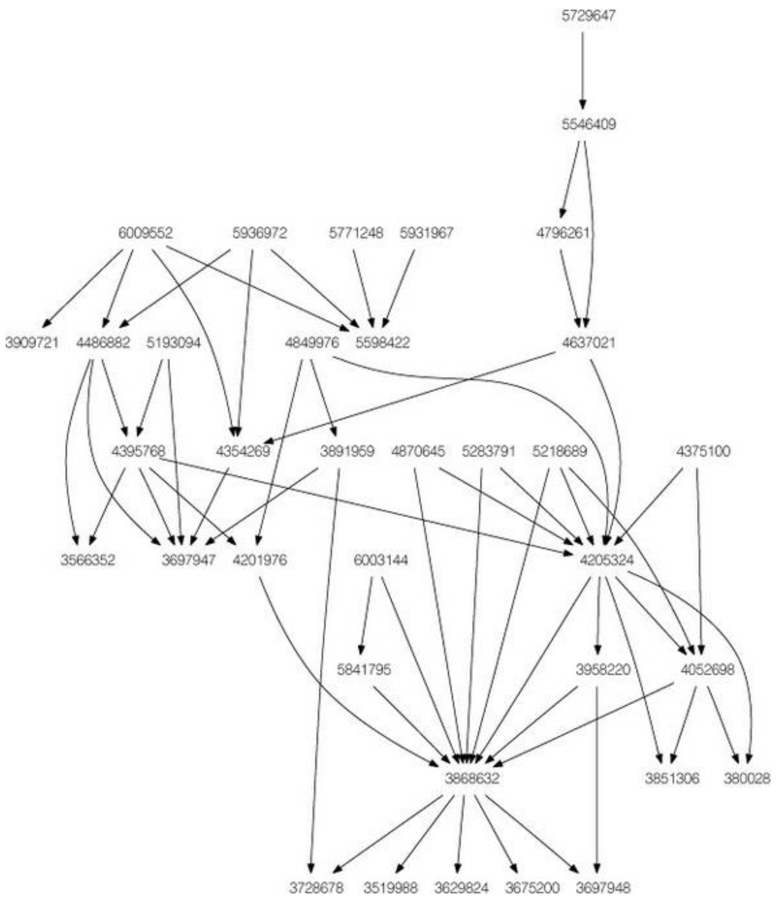


Figure 5.1 A partial view of the patent citation data set as a graph. Each patent is shown as a vertex (node), and each citation is a directed edge (arrow).

5.1.2 The patent description data

The other data set we use is the patent description data. It has the patent number, the patent application year, the patent grant year, the number of claims, and other metadata about patents. Look at the first few lines of this data set. It's similar to a table in a relational database, but in CSV format. This data set has more than 2.9 million records. As in many real-world data sets, it has many missing values.

```
"PATENT","GYEAR","GDATE","APPYEAR","COUNTRY","POSTATE","ASSIGNEE",
➥ "ASSCODE","CLAIMS","NCLASS","CAT","SUBCAT","CMADE","CRECEIVE",
➥ "RATIOCIT","GENERAL","ORIGINAL","FWDAPLAG","BCKGTLAG","SELFCTUB",
➥ "SELFCTLB","SECDUPBD","SECDLWBD"
3070801,1963,1096,,"BE","",,1,,269,6,69,,1,,0,,,,,,,
3070802,1963,1096,,"US","TX",,1,,2,6,63,,0,,,,,,,,,
3070803,1963,1096,,"US","IL",,1,,2,6,63,,9,,0.3704,,,,,,
3070804,1963,1096,,"US","OH",,1,,2,6,63,,3,,0.6667,,,,,,
3070805,1963,1096,,"US","CA",,1,,2,6,63,,1,,0,,,,,,
```

On the other hand, more recent patents are cited less often because only newer patents can be aware of their existence.

```
3070806,1963,1096,,"US","PA",,1,,2,6,63,,0,,,,,,,,
3070807,1963,1096,,"US","OH",,1,,623,3,39,,3,,0.4444,,,,,,
3070808,1963,1096,,"US","IA",,1,,623,3,39,,4,,0.375,,,,,,
3070809,1963,1096,,"US","AZ",,1,,4,6,65,,0,,,,,,,,
```

The first row contains the name of a couple dozen attributes, which are meaningful only to patent specialists. Even without understanding all the attributes, it's still useful to have some idea of a few of them. Table 4.1 describes the first ten.

Table 5.1 Definition of the first 10 attributes in the patent description data set

Attribute name	**Content**
`PATENT`	Patent number
`GYEAR`	Grant year
`GDATE`	Grant date, given as the number of days elapsed since January 1, 1960
`APPYEAR`	Application year (available only for patents granted since 1967) Country of first inventor
`COUNTRY`	Country of first inventor
`POSTATE`	State of first inventory (if country is U.S.)
`ASSIGNEE`	Numeric identifier for assignee (i.e., patent owner)
`ASSCODE`	One-digit (1-9) assignee type. (The assignee type includes U.S. individual, U.S. government, U.S. organization, non-U.S. individual, etc.)
`CLAIMS`	Number of claims (available only for patents granted since 1975)
`NCLASS`	3-digit main patent class

Now that we have two patent data sets, let's write Hadoop programs to process the data.

5.2 Constructing the basic template of a MapReduce program

We write most MapReduce programs in brief and as variations on a template. When writing a new MapReduce program, you generally take an existing MapReduce program and modify it until it does what you want. In this section, we write our first MapReduce program and explain its different parts. This program can serve as a template for future MapReduce programs.

Our first program will take the patent citation data and invert it. For each patent, we want to find and group the patents that cite it. Our output should be similar to the following:

```
1         3964859,4647229
10000     4539112
100000    5031388
1000006   4714284
1000007   4766693
1000011   5033339
1000017   3908629
1000026   4043055
1000033   4190903,4975983
1000043   4091523
1000044   4082383,4055371
1000045   4290571
1000046   5918892,5525001
1000049   5996916
1000051   4541310
1000054   4946631
1000065   4748968
1000067   5312208,4944640,5071294
1000070   4928425,5009029
```

We have discovered that patents 5312208, 4944640, and 5071294 cited patent 1000067. For this section we won't focus too much on the MapReduce data flow, which we've already covered in chapter 3. Instead we focus on the structure of a MapReduce program—in this case a Hadoop MapReduce v2 program. We need only one file for the entire program as you can see in listing 4.1.

Listing 5.1 Template for a typical Hadoop program

```
public class MyJob extends Configured implements Tool {

     public static class MapClass extends Mapper<Text, Text, Text, Text> {

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

context.write(value, key);
            }
         }

         public static class Reduce extends Reducer<Text, Text, Text, Text>
    {
             public void reduce(Text key, Iterator<Text> values,
                           Context context) throws IOException,
                               InterruptedException {

                    String csv = "";
                    while (values.hasNext()) {
                           if (csv.length() > 0) csv += ",";
                           csv += values.next().toString();
                    }
                    context.write(key, new Text(csv));
              }
        }

        public int run(String[] args) throws Exception {

              Configuration conf = getConf();
              Job job = Job.getInstance(conf, "patent");

              Path in = new Path(args[0]);
              Path out = new Path(args[1]);

              FileInputFormat.addInputPath(job, in);
              FileOutputFormat.setOutputPath(job, out);

              job.setJobName("MyJob");
              job.setMapperClass(MapClass.class);
              job.setReducerClass(Reduce.class);
              job.setInputFormatClass(KeyValueTextInputFormat.class);
              job.setOutputFormatClass(TextOutputFormat.class);
              job.setOutputKeyClass(Text.class);
              job.setOutputValueClass(Text.class);
              job.getConfiguration().set("key.value.separator.in.input.line", ",");

              System.exit(job.waitForCompletion(true) ? 0 : 1);
              return 0;
        }

        public static void main(String[] args) throws Exception {
              int res = ToolRunner.run(new Configuration(), new MyJob(), args);
              System.exit(res);
        }
}
```

Our convention is that a single class, called `MyJob` in this case, completely defines each MapReduce job. Hadoop requires the `Mapper` and the `Reducer` to be their own static classes. These classes are quite small, and our template includes them as inner classes to the `MyJob` class. The advantage is that everything fits in one file, simplifying code management. But keep in mind that these inner classes are independent and don't interact much with the `MyJob` class. Various nodes with different JVMs clone and run the `Mapper` and the `Reducer` during job execution, whereas the rest of the job class is executed only at the client machine.

We investigate the `Mapper` and the `Reducer` classes in a while. Without those classes, the skeleton of the `MyJob` class is

```
public class MyJob extends Configured implements Tool {

    public int run(String[] args) throws Exception {

        Configuration conf = getConf();

        Job job = Job.getInstance(conf, "patent");

        Path in = new Path(args[0]);
        Path out = new Path(args[1]);

          FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setJobName("MyJob");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormat(KeyValueTextInputFormat.class);
        job.setOutputFormat(TextOutputFormat.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);
        job.set("key.value.separator.in.input.line", ",");

        System.exit(job.waitForCompletion(true) ? 0 : 1);
        return 0;
     }

     public static void main(String[] args) throws Exception {

int res = ToolRunner.run(new Configuration(), new MyJob(), args);

        System.exit(res);
     }
}
```

The core of the skeleton is within the `run()` method, also known as the *driver*. The driver instantiates and configures a `Job` object named `job` that then executes using the `job.waitForCompletion()` function to start the MapReduce job and then wait until it completes. (The `JobClient` class, in turn, will communicate with the JobTracker to start the job across the cluster.) The `Job` object will hold all configuration parameters necessary for the job to run. The driver needs to pass to job the input paths, the output paths, the `Mapper` class, and the `Reducer` class—the basic parameters for every job. In addition, each job can reset the default job properties, such as `InputFormat`, `OutputFormat`, and so on. One can also call the `set()` method on the `JobConf` object to set up any configuration parameter—in our case we specify a string parameter for a variable `key.value.separator.in.input.line` that will be used by the Map class. Once you run the job, the configuration parameters are the master plan for the job. It becomes the blueprint for how the job will be run.

The `Job` object has many parameters, but we don't want to program the driver to set up all of them. The configuration files of the Hadoop installation are a good starting point. When starting a job from the command line, the user may also want to pass extra arguments to alter the job configuration. The driver can define its own set of commands and process the user arguments itself to enable the user to modify some of the configuration parameters. As this task is needed often, the Hadoop framework provides `ToolRunner`, `Tool`, and `Configured` to simplify it. When used together in the `MyJob` skeleton above, these classes enable our job to understand user-supplied options that are supported by `GenericOptionsParser`. For example, we have previously executed the `MyJob` class using this command line:

```
bin/hadoop jar playground/MyJob.jar MyJob input/cite75_99.txt output
```

Had we wanted to run the job only to see the mapper's output (which you may want to do for debugging purposes), we could set the number of reducers to zero with the option `-D mapred.reduce.tasks=0`.

```
bin/hadoop jar playground/MyJob.jar MyJob -Dmapred.reduce.tasks=0 ➥
       input/cite75_99.txt output
```

It works even though our program doesn't explicitly understand the -D option. By using ToolRunner, MyJob will automatically support the options in table 4.2.

Table 5.2 Options supported by GenericOptionsParser

Option	Description
`-conf <configuration file>`	Specify a configuration file.
`-D <property=value>`	Set value for a JobConf property.
`-fs <local|namenode:port>`	Specify a NameNode, can be "local".
`-jt <local|jobtracker:port>`	Specify a JobTracker.
`-files <list of files>`	Specify a comma-separated list of files to be used with the MapReduce job. These files are automatically distributed to all task nodes to be locally available.
`-libjars <list of jars>`	Specify a comma-separated list of jar files to be included in the classpath of all task JVMs.
`-archives <list of archives>`	Specify a comma-separated list of archives to be unarchived on all task nodes.

The convention for our template is to call the Mapper class MapClass and the Reducer class Reduce. The naming would seem more symmetrical if we call the Mapper class Map, but Java already has a class (interface) named Map. The Mapper and the Reducer extend Mapper and Reducer, respectively, which are small classes providing no-op implementations to the configure() and close() methods required by the two interfaces. We use the configure() and close() methods to set up and clean up the map (reduce) tasks. We won't need to override them except for more advanced jobs.

The signatures for the Mapper class and the Reducer class are

```
public static class MapClass extends Mappper<K1, V1, K2, V2> {

     public void map(K1 key, V1 value,
                    Context context) throws
                    IOException, InterruptedException { }
}

public static class Reduce extends Reducer<K2, V2, K3, V3> {

    public void reduce(K2 key, Iterator<V2> values, Context context)
              throws IOException, InterruptedException { }
}
```

The center of action for the Mapper class is the map() method and for the Reducer class the reduce() method. Each invocation of the map() method is given a key/value pair of types K1 and V1, respectively. The key/value pairs generated by the mapper are outputted via the write() method of the Context object. Somewhere in your map() method you need to call

```
context.write((K2) k, (V2) v);
```

Each invocation of the `reduce()` method at the reducer is given a key of type `K2` and an `Iterator` on a list of values of type `V2`. Note that it must be the same `K2` and `V2` types used in the `Mapper`. The `reduce()` method will likely have a loop to go through all the values of type `V2`.

```
while (values.hasNext()) { V2? v = values.next();
...
}
```

The `reduce()` method is also given an `Context` to gather its key/value output, which is of type `K3`/`V3`. Somewhere in the `reduce()` method you'll call

```
context.write((K3) k, (V3) v);
```

In addition to having consistent `K2` and `V2` types across `Mapper` and `Reducer`, you'll also need to ensure that the key and value types used in `Mapper` and `Reducer` are consistent with the input format, output key class, and output value class set in the driver. The use of `KeyValueTextInputFormat` means that `K1` and `V1` must both be type `Text`. The driver must call `setOutputKeyClass()` and `setOutputValueClass()` with the classes of `K2` and `V2`, respectively.

Finally, all the key and value types must be subtypes of `Writable`, which ensures a serialization interface for Hadoop to send the data around in a distributed cluster. In fact, the key types implement `WritableComparable`, a subinterface of Writable. The key types need to additionally support the `compareTo()` method, as keys are used for sorting in various places in the MapReduce framework.

5.2.1 MapReduce v1 and v2

In the code that we developed above, the interfaces were all the MapReduce version 2 or v2 forms. Hadoop 2 can accommodate both the v2 interfaces as well as code developed using the v1 interfaces. The version 2 interfaces slightly simplify programming the MapReduce model. For instance, the `Mapper` class just extends `Mapper` in v2 while it extended `MapReduceBase` and implemented `Mapper` in v1. The map and reduce function signatures similarly changed to use different classes for outputting results and reporting progress. Finally, configuring and running jobs changed slightly with the abandonment of the `JobConf` class.

Despite the changes, it is still possible to run most legacy v1 compiled code in Hadoop 2, and to use the v1 programming model as well. Compiling v1 code just requires that the appropriate include files (mapred for v1 instead of mapreduce for v2) are included in the source code.

5.3 Counting things

Much of what the layperson thinks of as statistics is counting, and many basic Hadoop jobs involve counting. We've already seen the word count example in chapter 1. For the patent citation data, we may want the number of citations a patent has received. This too is counting. The desired output would look like this:

```
1       2
10000   1
100000 1
1000006 1
1000007 1
1000011 1
1000017 1
1000026 1
1000033 2
1000043 1
1000044 2
1000045 1
1000046 2
1000049 1
1000051 1
1000054 1
1000065 1
1000067 3
```

In each record, a patent number is associated with the number of citations it has received. We can write a MapReduce program for this task. Like we said earlier, you hardly ever write a MapReduce program from scratch. You have an existing MapReduce program that processes the data in a similar way. You copy that and modify it until it fits what you want.

We already have a program for getting the inverted citation index. We can modify that program to output the count instead of the list of citing patents. We need the modification only at the `Reducer`. If we choose to output the count as an `IntWritable`, we need to specify `IntWritable` in three places in the `Reducer` code. We called them `V3` in our previous notation.

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

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

        int count = 0;
        while (values.hasNext()) {
            values.next();
            count++;
        }
        context.write(key, new IntWritable(count));
    }
}
```

By changing a few lines and matching class types, we have a new MapReduce program. This program may seem a minor modification. Let’s go through another example that requires more changes, but you’ll see that the basic MapReduce program structure remains.

After running the previous example, we now have a data set that counts the number of citations for each patent. A neat exercise would be to count the counts. Let’s build a histogram of the citation counts. We expect a large number of patents to have been cited only once, and a small number may have been cited hundreds of times. It would be interesting to see the distribution of the citation counts.

NOTE As the patent citation data set only covers patents issued between 1975 and 1999, the citation count is necessarily an underestimate. (Citations from patents outside of that period aren't counted.) We also don't deal with patents that supposedly have been cited zero times. Despite these caveats, the analysis will be useful.

The first step to writing a MapReduce program is to figure out the data flow. In this case, as a mapper reads a record, it ignores the patent number and outputs an intermediate key/value pair of `<citation_count, 1>`. The reducer will sum up the number of 1s for each citation count and output the total.

After figuring out the data flow, decide on the types for the key/value pairs—`K1`, `V1`, `K2`, `V2`, `K3`, and `V3` for the input, intermediate, and output key/value pairs. Let's use the `KeyValueTextInputFormat`, which automatically breaks each input record into key/value pairs based on a separator character. The input format produces `K1` and `V1` as Text. We choose to use `IntWritable` for `K2`, `V2`, `K3`, and `V3` because we know those data must be integers and it's more efficient to use `IntWritable`.

Based on the data flow and the data types, you'll be able to see the final program shown in listing 4.2 and understand what it's doing. You can see that it's structurally similar to the other MapReduce programs we've seen so far. We go into details about the program after the listing.

Listing 5.2 CitationHistogram.java: count patents cited once, twice, and so on

```
public class CitationHistogram extends Configured implements Tool {

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

        private final static IntWritable uno = new IntWritable(1);
        private IntWritable citationCount = new IntWritable();

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

            citationCount.set(Integer.parseInt(value.toString()));
            context.write(citationCount, uno);
        }
    }

    public static class Reduce extends Reducer<IntWritable,IntWritable,
IntWritable,IntWritable>
    {

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

            int count = 0;
            while (values.hasNext()) {
                count += values.next().get();
            }
            context.write(key, new IntWritable(count));
        }
    }

    public int run(String[] args) throws Exception {
        Configuration conf = getConf();

        Job job = Job.getInstance(conf, "histogram");

        Path in = new Path(args[0]);
        Path out = new Path(args[1]);
        FileInputFormat.setInputPaths(job, in);
        FileOutputFormat.setOutputPath(job, out);

        job.setJobName("CitationHistogram");
        job.setMapperClass(MapClass.class);
        job.setReducerClass(Reduce.class);

        job.setInputFormatClass(KeyValueTextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);

        return 0;
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(),
                                 new CitationHistogram(),
                                 args);

        System.exit(res);
    }
}
```

The class name is now `CitationHistogram`; all references to `MyJob` were changed to reflect the new name. The `main()` method is almost always the same. The driver is mostly intact. The input format and output format are still `KeyValueTextInputFormat` and `TextOutputFormat`, respectively. The main change is that the output key class and the output value class are now `IntWritable`, to reflect the new type for `K2` and `V2`. We've also removed this line:

```
job.set("key.value.separator.in.input.line", ",");
```

It sets the separator character used by `KeyValueTextInputFormat` to break each input line into a key/value pair. Previously it was a comma for processing the original patent citation data. By not setting this property it defaults to the tab character, which is appropriate for the citation count data.

The data flow for this mapper is similar to that of the previous mappers, only here we've chosen to define and use a couple of class variables—`citationCount` and `uno`.

```
public static class MapClass extends Mapper<Text, Text, IntWritable, IntWritable> {
    private final static IntWritable uno = new IntWritable(1);
    private IntWritable citationCount = new IntWritable();

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

            citationCount.set(Integer.parseInt(value.toString()));
            context.write(citationCount, uno);
        }
    }
```

The `map()` method has one extra line for setting `citationCount`, which is for type casting. The reason for defining `citationCount` and `uno` in the class rather than inside the method is purely one of efficiency. The `map()` method will be called as many times as there are records (in a split, for each JVM). Reducing the number of objects created inside the `map()` method can increase performance and reduce garbage collection. As we pass `citationCount` and `uno` to `context.write()`, we're relying on the `context.write()` method's contract to not modify those two objects.[10]

The reducer sums up the values for each key. It seems inefficient because we know all values are 1s (`uno`, to be exact). Why do we need to sum the count? We've chosen this route because it will be easier for us later if we choose to add a combiner to enhance performance. Unlike `MapClass`, the call to `context.write()` in `Reduce` instantiates a new `IntWritable` rather than reuse an existing one.

```
collect.write(key, new IntWritable(count));
```

We can improve performance by using an `IntWritable` class variable. But the number of times `reduce()` is called is much smaller in this particular program, probably no more than a thousand times (across all reducers). We don't have much need to optimize this particular code.

Running the MapReduce job on the citation count data will show the following result. As we suspect, a large number (900K+) of patents have only one citation, whereas some have hundreds of citations. The most popular patent has 779 citations.

```
1      921128
2      552246
3      380319
4      278438
5      210814
6      163149
7      127941
8      102155
9      82126
10     66634
```

[10] We see in section 5.1.3 that this reliance will forbid the `ChainMapper` from using pass-by-reference.

...

```
411    1
605    1
613    1
631    1
633    1
654    1
658    1
678    1
716    1
779    1
```

As this histogram output is only several hundred lines long, we can put it into a spreadsheet and plot it. Figure 4.2 shows the number of patents at various citation frequencies. The plot is on a log-log scale. When a distribution shows as a line in a log-log plot, it's considered to be a power law distribution. The citation count histogram seems to fit the description, although its approximately parabolic curvature also suggests a lognormal distribution.

As you've seen in our examples so far, a MapReduce program is often not very big, and you can keep a certain structure across them to simplify development. Most of the work is in thinking through the data flow.

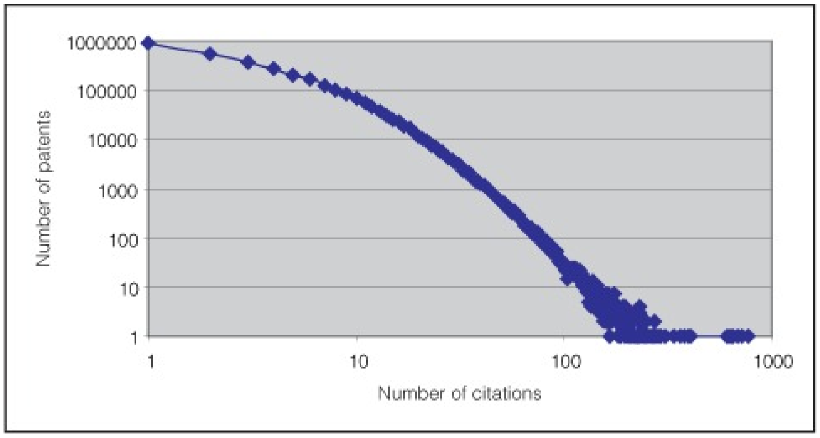


Figure 5.2 Plotting the number of patents at different citation frequencies. Many patents have one citation (or not at all, which is not shown on this graph). Some patents have hundreds of citations. On a log-log graph, this looks close enough to a straight line to be considered a power-law distribution.

5.4 Streaming in Hadoop

We have been using Java to write all our Hadoop programs. Hadoop supports other languages via a generic API called Streaming. In practice, Streaming is most useful for writing simple, short MapReduce programs that are more rapidly developed in a scripting language that can take advantage of non-Java libraries.

Hadoop Streaming interacts with programs using the Unix streaming paradigm. Inputs come in through STDIN and outputs go to STDOUT. Data has to be text based and each line is considered a record. Note that this is exactly how many Unix commands work, and Hadoop Streaming enables those commands to be used as mappers and reducers. If you're familiar with using Unix commands, such as `wc`, `cut`, or `uniq` for data processing, you can apply them to large data sets using Hadoop Streaming.

The overall data flow in Hadoop Streaming is like a pipe where data streams through the mapper, the output of which is sorted and streamed through the reducer. In pseudo-code using Unix's command line notation, it's

```
cat [input_file] | [mapper] | sort | [reducer] > [output_file]
```

The following examples will illustrate how to use Streaming with Unix commands.

5.4.1 Streaming with Unix commands

In the first example, let's get a list of cited patents in cite75_99.txt.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -input input/cite75_99.txt
➥ -output output
➥ -mapper 'cut -f 2 -d ,'
➥ -reducer 'uniq'
```

That's it! It's a one-line command. Let's see what each part of the command does.

The Streaming API is in a `contrib` package at contrib/streaming/hadoop-*streaming.jar. The first part and the `-input` and the `-output` arguments specify that we're running a Streaming program with the corresponding input and output file/directory. The mapper and reducer are specified as arguments in quotes. We see that for the mapper we use the Unix `cut` command to extract the second column, where columns are separated by commas. In the citation data set this column is the patent number of a cited patent. These patent numbers are then sorted and passed to the reducer. The `uniq` command at the reducer will remove all duplicates in the sorted data. The output of this command is

```
"CITED"
1
10000
100000
1000006
...
999973
999974
999977
999978
999983
```

The first row has the column descriptor "CITED" from the original file. Note that the rows are sorted lexicographically because Streaming processes everything as text and doesn't know other data types.

After getting the list of cited patents, we may want to know how many are there. Again we can use Streaming to quickly get a count, using the Unix command `wc -l`.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -D mapred.reduce.tasks=0
➥ -input output
➥ -output output_a
➥ -mapper 'wc -l'
```

Here we use `wc -l` as the mapper to count the number of records in each split. Hadoop Streaming (since version 0.19.0) supports the `GenericOptionsParser`. The `-D` argument is used for specifying configuration properties. We want the mapper to directly output the record count without any reducer, so we set `mapred.reduce.tasks` to 0 and don't specify the `-reducer` option at all.[11] The final count is 3258984. More than 3 million patents have been cited according to our data.

5.4.2 Streaming with scripts

We can use any executable script that processes a line-oriented data stream from STDIN and outputs to STDOUT with Hadoop Streaming. For example, the Python script in listing 4.4 randomly samples data from STDIN. For those who don't know Python, the program has a `for` loop that reads STDIN one line at a time. For each line, we choose a random integer between 1 and 100 and check against the user-given argument (`sys.argv[1]`). The comparison determines whether to pass that line on to the output or ignore it. You can use the script in Unix to uniformly sample a line-oriented data file, for example:

```
cat input.txt | RandomSample.py 10 > sampled_output.txt
```

The preceding command calls the Python script with an argument of 10; sampled_output.txt will have (approximately) 10 percent of the records in input.txt. We can in fact specify any integer between 1 and 100 to get the corresponding percentage of data in the output.

Listing 5.4 RandomSample.py: a Python script printing random lines from StdIn

```
#!/usr/bin/env python

import sys, random

for line in sys.stdin:
    if (random.randint(1,100) <= int(sys.argv[1])):
        print line.strip()
```

We can apply the same script in Hadoop to get a smaller sample of a data set. A sampled data set is often useful for development purposes, as you can run your Hadoop program on the

[11] You may notice that this approach counts the number of records in each split, not the entire file. With a bigger file, or multiple files, the user will have to sum up the counts herself to get the overall total. To fully automate a complete counting, the user will have to write a script at the reducer to sum up all the partial counts.

sampled data in standalone or pseudo-distributed mode to quickly debug and iterate. Also, when you're looking for some "descriptive" information about your data, the speed and convenience in processing a smaller data set generally outweigh any loss of precision. Finding data clusters is one example of such descriptive information. Optimized implementations of a variety of clustering algorithms are readily available in R, MATLAB, and other packages. It makes a lot more sense to sample down the data and apply some standard software package, instead of trying to process all data using some distributed clustering algorithms in Hadoop.

WARNING The loss of precision from computing on a sampled data set may or may not be important. It depends on what you're trying to compute and the distribution of your data set. For example, it's usually fine to compute an average from a sampled data set, but if the data set is highly skewed and the average is dominated by a few values, sampling can be problematic. Similarly, clustering on a sampled data set is fine if it's used only to get a general understanding of the data. If you were looking for small, anomalous clusters, sampling may get rid of them. For functions such as `maximum` and `minimum`, it's not a good idea to apply them to sampled data.

Running RandomSample.py using Streaming is like running Unix commands using Streaming, the difference being that Unix commands are already available on all nodes in the cluster, whereas RandomSample.py is not. Hadoop Streaming supports a `-file` option to package your executable file as part of the job submission.[12] Our command to execute RandomSample.py is

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -D mapred.reduce.tasks=1
➥ -input input/cite75_99.txt
➥ -output output
➥ -mapper 'RandomSample.py 10'
➥ -file RandomSample.py
```

In specifying the mapper to be `'RandomSample.py 10'` we're sampling at 10 percent. Note that we've set the number of reducers (`mapred.reduce.tasks`) to 1. As we haven't specified any particular reducer, it will use the default `IdentityReducer`. As its name implies, `IdentityReducer` passes its input straight to output. In this case we can set the number of reducers to any non-zero value to get an exact number of output files. Alternatively, we can set the number of reducers to 0, and let the number of output files be the number of mappers. This is probably not ideal for the sampling task as each mapper's output is only a small fraction of the input, and we may end up with a number of small files. We can easily correct that later using the HDFS shell command `getmerge` or other file manipulations to arrive at the right number of output files. The approach to use is more or less a personal preference.

[12] It's also implicitly assumed that you have installed the Python language on all the nodes in your cluster.

The random sampling script was implemented in Python, although any scripting language that works with STDIN and STDOUT would work. For illustration we've rewritten the same script in PHP[13] (listing 4.5). Execute this Stream script with

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -D mapred.reduce.tasks=1
➥ -input input/cite75_99.txt
➥ -output output
➥ -mapper 'php RandomSample.php 10'
➥ -file RandomSample.php
```

Listing 5.5 RandomSample.php.: a PHP script printing random lines from StDIn

```
<?php

while (!feof(STDIN)) {
    $line = fgets(STDIN);
    if (mt_rand(1,100) <= $argv[1]) {
        echo $line;
    }
}
```

The random sampling scripts don't require a custom reducer, but you can't always write a Streaming program like that. As you'll use Streaming quite often in practice, let's see another exercise. This time we create a custom reducer.

Suppose we're interested in finding the most number of claims in a single patent. In the patent description data set, the number of claims for a given patent is in the ninth column. Our task is to find the maximum value in the ninth column of the patent description data.

Under Streaming, each mapper sees the entire stream of data, and it's the mapper that takes on the responsibility of breaking the stream into (line-oriented) records. In the standard Java model, the framework itself breaks input data into records, and gives the `map()` method only one record at a time. The Streaming model makes it easy to keep state information across records in a split, which we take advantage of in computing the maximum. The standard Java model, too, can keep track of state across records in a split, but it's more involved. We cover that in the next chapter.

In creating a Hadoop program for computing the maximum, we take advantage of the distributive property of maximum. Given a data set divided into many splits, the global maximum is the maximum over the maxima of the splits. That sounded like a mouthful, but a

[13] You may have noticed in listing 4.5 that there's no ending bracket ?> to close the opening bracket <?php. Recall that PHP was originally designed to work within static HTML content. Anything outside the PHP brackets <?php ... ?> is considered static content to be outputted. When using PHP as a pure scripting language, you need to be careful that you leave no whitespaces outside the brackets. Otherwise they will be outputted and may cause unintended behavior that is hard to debug. (It would appear whitespaces were introduced in the output data out of nowhere.)
It's easy to ensure that there's no whitespaces before the opening bracket <?php by putting the bracket at the beginning of the script file. But, it's easy to accidentally leave whitespaces after the closing bracket ?>, as ending whitespaces don't grab attention. When using a file as a PHP script, it's safer to omit the closing bracket ?>. The PHP interpreter will quietly read everything till the end-of-file as PHP commands rather than static content.

simple example will make it clear. If we have four records X1, X2, X3, and X4, and they're divided into two splits (X1, X2) and (X3, X4), we can find the maximum over all four records by looking at the maximum of each split, or

```
max(X1,X2,X3,X4) = max(max(X1,X2), max(X3,X4))
```

Our strategy is to have mapper calculate the maximum over its individual split. Each mapper will output a single value at the end. We have a single reducer that looks at all those values and outputs the global maximum. Listing 4.6 depicts the Python script for a mapper to compute the maximum over a split.

Listing 5.6 AttributeMax.py: Python script to find maximum value of an attribute

```
#!/usr/bin/env python

import sys

index = int(sys.argv[1])
max   = 0
for line in sys.stdin:
    fields = line.strip().split(",")
    if fields[index].isdigit():
        val = int(fields[index])
        if (val > max):
            max = val
else:
    print max
```

The script is not complicated. It has a `for` loop to read one record at a time. It tokenizes the record into fields and updates the maximum if the user-specified field is bigger. Note that the mapper doesn't output any value until the end, when it sends out the maximum value of the entire split. This is different from what we've seen before, where each record sends out one or more intermediate records to be processed by the reducers.

Given the parsimonious output of the mapper, we can use the default `IdentityReducer` to record the (sorted) output of the mappers.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -D mapred.reduce.tasks=1
➥ -input input/apat63_99.txt
➥ -output output
➥ -mapper 'AttributeMax.py 8'
➥ -file playground/AttributeMax.py
```

The mapper is `'AttributeMax.py 8'`. It outputs the maximum of the ninth column in a split. The single reducer collects all the mapper outputs. Given seven mappers, the final output of the above command is this:

```
0
260
306

348
```

```
394
706
868
```

Each line records the maximum over a particular split. We see that one split has zero claims in all its records. This sounds suspicious until we recall that the claim count attribute is not available for patents before 1975.

We see that our mapper is doing the right thing. We can use a reducer that outputs the maximum over the values outputted by the mappers. We have an interesting situation here, due to the distributive property of maximum, where we can also use AttributeMax.py as the reducer. Only now the reducer is trying to find the maximum in the "first" column.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -D mapred.reduce.tasks=1
➥ -input input/apat63_99.txt
➥ -output output
➥ -mapper 'AttributeMax.py 8'
➥ -reducer 'AttributeMax.py 0'
➥ -file AttributeMax.py
```

The output of the above command should be a one-line file, and you'll find the maximum number of claims in a patent to be 868.

Classes of aggregation functions

We use aggregation functions to compute descriptive statistics. They're generally grouped into three classes: distributive, algebraic, and holistic. The maximum function is an example of a distributive function. Other distributive functions include minimum, sum, and count. As the name implies, distributive functions have distributive properties. Similar to the maximum function, you can globally compute these functions by iteratively applying them to smaller chunks of data.

Another class of aggregation functions is the algebraic functions. Examples of this class include average and variance. They don't follow the distributive property, and their derivation will require some "algebraic" computation over simpler functions. We get into examples of this in the next section.

Finally, functions such as `median` and `K` smallest/largest value belong to the holistic class of aggregation functions. Readers interested in a challenge should try to implement the median function in an efficient manner using Hadoop.

5.4.3 Streaming with key/value pairs

At this point you may wonder what happened to the key/value pair way of encoding records. Our discussion on Streaming so far talks about each record as an atomic unit rather than as composed of a key and a value. The truth is that Streaming works on key/value pairs just like the standard Java MapReduce model. By default, Streaming uses the tab character to separate the key from the value in a record. When there's no tab character, the entire record is considered the key and the value is empty text. For our data sets, which have no tab character, this provides the illusion that we're processing each individual record as a whole unit. Furthermore, even if the records do have tab characters in them, the Streaming API will only shuffle and sort the records in a different order. As long as our mapper and reducer work in a record-oriented way, we can maintain the record-oriented illusion.

Working with key/value pairs allows us to take advantage of the key-based shuffling and sorting to create interesting data analyses. To illustrate key/value pair processing using Streaming, we can write a program to find the maximum number of claims in a patent for each country. This would differ from AttributeMax.py in that this is trying to find the maximum for each key, rather than a maximum across all records. Let's make this exercise more interesting by computing the average rather than finding the maximum. (As we see, Hadoop already includes a package called Aggregate that contains classes that help find the maximum for each key.)

First, let's examine how key/value pairs work in the Streaming API for each step of the MapReduce data flow.

1. As we've seen, the mapper under Streaming reads a split through STDIN and extracts each line as a record. Your mapper can choose to interpret each input record as a key/value pair or a line of text.
2. The Streaming API will interpret each line of your mapper's output as a key/ value pair separated by tab. Similar to the standard MapReduce model, we apply the partitioner to the key to find the right reducer to shuffle the record to. All key/value pairs with the same key will end up at the same reducer.
3. At each reducer, key/value pairs are sorted according to the key by the Streaming API. Recall that in the Java model, all key/value pairs of the same key are grouped together into one key and a list of values. This group is then presented to the `reduce()` method. Under the Streaming API your reducer is responsible for performing the grouping. This is not too bad as the key/value pairs are already sorted by key. All records of the same key are in one contiguous chunk. Your reducer will read one line at a time from STDIN and will keep track of the new keys.
4. For all practical purposes, the output (STDOUT) of your reducer is written to a file directly. Technically a no-op step is taken before the file write. In this step the Streaming API breaks each line of the reducer's output by the tab character and feeds the key/value pair to the default `TextOutputFormat`, which by default re-inserts the tab character before writing the result to a file. Without tab characters in the reducer's output it will show the same no-op behavior. You can reconfigure the default behavior to do something different, but it makes sense to leave it as a no-op and push the processing into your reducer.

To understand the data flow better, we write a Streaming program to compute the average number of claims for each country. The mapper will extract the country and the claims count for each patent and package them as a key/value pair. In accord with the default Streaming convention, the mapper outputs this key/value pair with a tab character to separate them. The Streaming API will pick up the key and the shuffling will guarantee that all claim counts of a country will end up at the same reducer. The Python code for this is shown in listing 4.7. For each record, the mapper extracts the country (`fields[4][1:-1]`) as key and the claims count (`fields[8]`) as value. An extra concern with our data set is that there may be missing values. We've added a conditional statement to skip over records with missing claim counts.

Listing 5.7 AverageByAttributeMapper.py: output country and claim count of patents

```
#!/usr/bin/env python

import sys

for line in sys.stdin:
    fields = line.split(",")
    if (fields[8] and fields[8].isdigit()):
        print fields[4][1:-1] + "\t" + fields[8]
```

Before writing the reducer, let's run the mapper in two situations: without any reducer, and with the default `IdentityReducer`. It's a useful approach for learning as we can see exactly what's being outputted by the mapper (by using no reducer) and what's being inputted into the reducer (by using `IdentityReducer`). You'll find this handy later when debugging your MapReduce programs. Using this approach, you can at least check if the mapper is outputting the proper data and if the proper data is being sent to the reducer. First let's run the mapper without any reducer.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -D mapred.reduce.tasks=0
➥ -input input/apat63_99.txt
➥ output output
➥ file AverageByAttributeMapper.py
➥ mapper 'AverageByAttributeMapper.py'
```

The output should consist of lines where a country code is followed by a tab followed by a numeric count. The order of the output records is not sorted by (the new) key. In fact, it's in the same order as the order of the input records, although that's not obvious from looking at the output.

The more interesting case is to use the IdentityReducer with a non-zero number of reducers. We see how the shuffled and sorted records are presented to the reducer. To keep it simple let's try a single reducer by setting `-D mapred.reduce.tasks=1` and see the first 32 records.

```
AD      9
AD      12
AD      7
AD      28
AD      14
AE      20
AE      7
AE      35
AE      11
AE      12
AE      24
AE      4
AE      16
AE      26
AE      11
AE      4
AE      23
AE      12
AE      16
AE      10
AG      18
AG      12
AG      8
AG      14
AG      24
AG      20
AG      7
AG      3
AI      10
AM      18
AN      5
AN      26
```

Under the Streaming API, the reducer will see these text data in STDIN. We have to code our reducer to recover the key/value pairs by breaking each line at the tab character. Sorting has "grouped" together records of the same key. As you read each line from STDIN, you'll be responsible for keeping track of the boundary between records of different keys. Note that although the keys are sorted, the values don't follow any particular order. Finally, the reducer must perform its stated computation, which in this case is calculating the average value across a key. Listing 4.8 gives the complete reducer in Python.

Listing 5.8 AverageByAttributeReducer.py

```
#!/usr/bin/env python

import sys

 (last_key, sum, count) = (None, 0.0, 0)

for line in sys.stdin:
    (key, val) = line.split("\t")

    if last_key and last_key != key:
        print last_key + "\t" + str(sum / count)
        (sum, count) = (0.0, 0)

    last_key = key
    sum   += float(val)
    count += 1

print last_key + "\t" + str(sum / count)
```

The program keeps a running sum and count for each key. When it detects a new key in the input stream or the end of the file, it computes the average for the previous key and sends it to STDOUT. After running the entire MapReduce job, we can easily check the correctness of the first few results.

```
AD     14.0
AE     15.4
AG     13.25
AI     10.0
AM     18.0
AN     9.625
```

NOTE For those interested, the NBER website from where we get the patent data also has a file (list_of_countries.txt) that shows the full country name for each country code. Looking at the output of our job and the country codes, we see that Andorra (AD) patents have an average 14 claims. Arab Emirates (AE) patents average 15.4 claims. Antigua and Barbuda (AG) patents average 13.25 claims, and so forth.

5.4.4 Streaming with the Aggregate package

Hadoop includes a library package called `Aggregate` that simplifies obtaining aggregate statistics of a data set. This package can simplify the writing of Java statistics collectors, especially when used with Streaming, which is the focus of this section.[14]

The Aggregate package under Streaming functions as a reducer that computes aggregate statistics. You only have to provide a mapper that processes records and sends out a specially formatted output. Each line of the mapper's output looks like

```
function:key\tvalue
```

The output string starts with the name of a value aggregator function (from the set of predefined functions available in the `Aggregate` package). A colon and a tab-separated key/value pair follows. The `Aggregate` reducer applies the function to the set of values for each key. For example, if the function is `LongValueSum`, then the output is the sum of values for each key. (As the function name implies, each value is treated as a Java long type.) If the function is `LongValueMax`, then the output is the maximum value for each key. You can see the list of aggregator functions supported in the Aggregate package in table 4.3.

Table 5.3 List of value aggregator functions supported by the `Aggregate` package

Value aggregator	Description
`DoubleValueSum`	Sums up a sequence of double values.
`LongValueMax`	Finds the maximum of a sequence of long values.
`LongValueMin`	Finds the minimum of a sequence of long values.
`LongValueSum`	Sums up a sequence of long values.

[14] Using the `Aggregate` package in Java is explained in http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/mapred/lib/aggregate/package-summary.html.

`StringValueMax`	Finds the lexicographical maximum of a sequence of string values.
`StringValueMin`	Finds the lexicographical minimum of a sequence of string values.
`UniqValueCount`	Finds the number of unique values (for each key).
`ValueHistogram`	Finds the count, minimum, median, maximum, average, and standard deviation of each value. (See text for further explanation.)

Let's go through an exercise using the `Aggregate` package to see how easy it is. We want to count the number of patents granted each year. We can approach this problem in a way similar to the word counting example we saw in chapter 1. For each record, our mapper will output the grant year as the key and a "1" as the value. The reducer will sum up all the values ("1"s) to arrive at a count. Only now we're using Streaming with the Aggregate package. Our result will be the simple mapper shown in listing 4.9.

Listing 5.9 AttributeCount.py

```
#!/usr/bin/env python

import sys

index = int(sys.argv[1])
for line in sys.stdin:
    fields = line.split(",")
    print "LongValueSum:" + fields[index] + "\t" + "1"
```

AttributeCount.py works for any CSV-formatted input file. The user only has to specify the column index to count the number of records for each attribute in that column. The `print` statement has the main "action" of this short program. It tells the `Aggregate` package to sum up all the values (of 1) for each key, defined as the user-specified column (index). To count the number of patents granted each year, we run this Streaming program with the `Aggregate` package, telling the mapper to use the second column (index = 1) of the input file as the attribute of interest.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -input input/apat63_99.txt
➥ -output output
➥ -file AttributeCount.py
➥ -mapper 'AttributeCount.py 1'
➥ -reducer aggregate
```

You'll find most of the options for running the Streaming program familiar. The main thing to point out is that we've specified the reducer to be `'aggregate'`. This is the signal to the Streaming API that we're using the `Aggregate` package. The output of the MapReduce job (after sorting) is

```
"GYEAR" 1
1963   45679
1964   47375
1965   62857
```

```
...
1996   109645
1997   111983
1998   147519
1999   153486
```

The first row is anomalous because the first row of the input data is a column description. Otherwise the MapReduce job neatly outputs the patent count for each year. As shown in figure 4.3, we can plot the data to visualize it better. You'll see that it has a mostly steady upward trend.

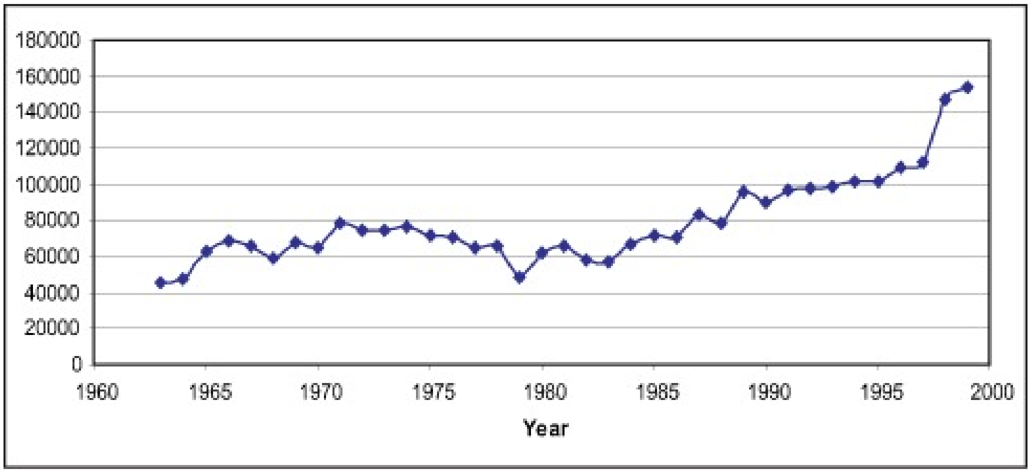


Figure 5.3 Using Hadoop to count patents published each year and Microsoft Excel to plot the result. this analysis using Hadoop quickly shows the annual patent output to have almost quadrupled in 40 years.

Looking at the list of functions in the `Aggregate` package in table 4.3, you'll find that most of them are combinations of maximum, minimum, and sum for atomic data types. (For some reason `DoubleValueMax` and `DoubleValueMin` aren't supported. They would be trivial modifications of `LongValueMax` and `LongValueMin`, and an added advantage.) `UniqValueCount` and `ValueHistogram` are slightly different and we look at some examples of how to use them.

`UniqValueCount` gives the number of unique values for each key. For example, we may want to know whether more countries are participating in the U.S. patent system over time. We can examine this by looking at the number of countries with patents granted each year. We use a straightforward wrapper of `UniqValueCount` in listing 4.10 and apply it to the year and country columns of apat63_99.txt (column index of 1 and 4, respectively).

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -input input/apat63_99.txt
➥ -output output
➥ -file UniqueCount.py
➥ -mapper 'UniqueCount.py 1 4'
➥ -reducer aggregate
```

In the output we get one record for each year. Plotting it gives us figure 4.4. We can see that the increasing number of patents granted from 1960 to 1990 (from figure 4.3) didn't come from more countries (figure 4.4). The same number of countries had filed more.

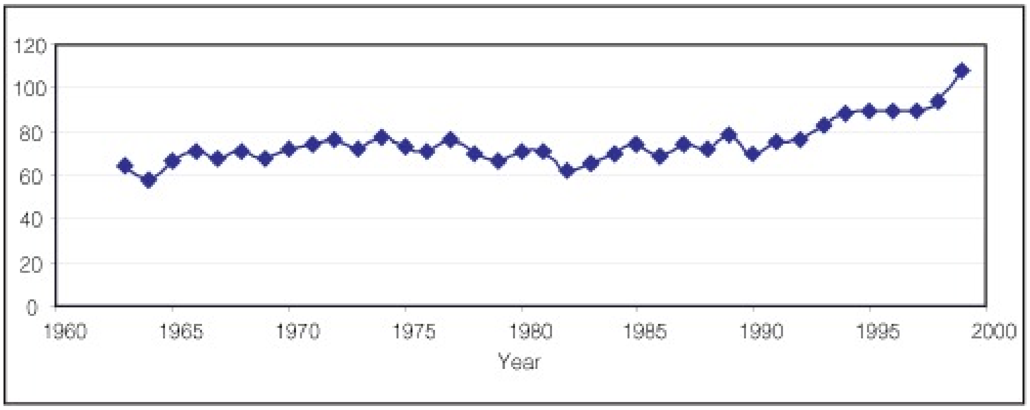


Figure 5.4 The number of countries with U.S. patents granted in each year. We performed the computation with a MapReduce job and graphed the result with Microsoft Excel.

Listing 5.10 UniqueCount.py: a wrapper around the UniqValueCount function

```
#!/usr/bin/env python

import sys

index1 = int(sys.argv[1]) index2 = int(sys.argv[2])

for line in sys.stdin:
    fields = line.split(",")
    print "UniqValueCount:" + fields[index1] + "\t" + fields[index2]
```

The aggregate function `ValueHistogram` is the most ambitious function in the `Aggregate` package. For each key, it outputs the following:

- The number of unique values
- The minimum count
- The median count
- The maximum count
- The average count
- The standard deviation

In its most general form, it expects the output of the mapper to have the form

```
ValueHistogram:key\tvalue\tcount
```

We specify the function `ValueHistogram` followed by a colon, followed by a tab-separated key, value, and count triplet. The `Aggregate` reducer outputs the six statistics above for each key. Note that for everything except the first statistics (number of unique values) the counts are summed over each key/value pair. Outputting two records from your mapper as

```
ValueHistogram:key_a\tvalue_a\t10
ValueHistogram:key_a\tvalue_a\t20
```

is no different than outputting a single record with the sum

```
ValueHistogram:key_a\tvalue_a\t30
```

A useful variation is for the mapper to only output the key and value, without the count and the tab character that goes with it. `ValueHistogram` automatically assumes a count of 1 in this case. Listing 4.11 shows a trivial wrapper around `ValueHistogram`.

Listing 5.11 ValueHistogram.py: wrapper around Aggregate package's ValueHistogram

```
#!/usr/bin/env python

import sys

index1 = int(sys.argv[1])
index2 = int(sys.argv[2])

for line in sys.stdin:
    fields = line.split(",")
    print "ValueHistogram:" + fields[index1] + "\t" + fields[index2]
```

We run this program to find the distribution of countries with patents granted for each year.

```
bin/hadoop jar share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar
➥ -input input/apat63_99.txt
➥ -output output
➥ -file ValueHistogram.py
➥ -mapper 'ValueHistogram.py 1 4'
➥ -reducer aggregate
```

The output is a tab-separated value (TSV) file with seven columns. The first column, the year of patent granted, is the key. The other six columns are the six statistics the `ValueHistogram` is set to compute. A partial view of the output is here (we skip the first two rows for formatting reasons):

```
1963   64        1        5        37174    713.734375       4610.076525402627
1964   58        1        7        38410    816.8103448275862        4997.413601595352
1965   67        1        5        50331    938.1641791044776        6104.779230296307
1966   71        1        5        54634    963.4507042253521        6443.625995189338
1967   68        1        8        51274    965.4705882352941        6177.445623039149
1968   71        1        7        45781    832.4507042253521        5401.229955880634
1969   68        1        8        50394    993.5147058823529        6080.713518728092
1970   72        1        7        47073    894.8472222222222        5527.883233761672
1971   74        1        9        55976    1058.337837837838        6492.837390992137
1972   76        1        7        51518    984.3421052631579        5916.340924557642
1973   72        1        10       51501    1029.763888888889        6066.751172701396
1974   77        1        8        50646    990.6233766233767        5794.207539776167
```

```
1975   73        1        9        46715    986.3013698630137        5501.2231458973765
1976   71        1        11       44280    989.0985915492957        5299.050345839762
1977   76        1        8        41490    858.8026315789474        4801.305653070173
1978   70        1        11       41254    944.3142857142857        4984.348799830059
1979   67        1        10       30078    729.1641791044776        3718.7885990853274
1980   71        1        11       37354    870.6901408450705        4504.845371317451
1981   71        1        10       39223    926.3521126760563        4752.435224825007
```

The first column after the year is the number of unique values. This is exactly the same as the output of `UniqValueCount`. The second, third, and fourth columns are the minimum, median, and maximum, respectively. For the patent data set we used, we see that (for every year) the country receiving the fewest granted patents (other than 0) received 1. Looking specifically at the output for 1964, the country receiving the most patents received 38410 patents, whereas half the countries received less than 7 patents. The average number of patents a country received in 1964 is 816.8 with a standard deviation of 4997.4. Needless to say, the number of patents granted to each country is highly skewed, given the discrepancy between the median (7) and the average (816.8).

We've seen how using the `Aggregate` package under Streaming is a simple way to get some popular metrics. It's a great demonstration of Hadoop's power in simplifying the analysis of large data sets.

5.5 Improving performance with combiners

We saw in AverageByAttributeMapper.py and AverageByAttributeReducer.py (listings 4.7 and 4.8) how to compute the average for each attribute. The mapper reads each record and outputs a key/value pair for the record's attribute and count. It shuffles the key/value pairs across the network, and the reducer computes the average for each key. In our example of computing the average number of claims for each country's patents, we see at least two efficiency bottlenecks:

- If we have 1 billion input records, the mappers will generate 1 billion key/value pairs that will be shuffled across the network. If we were computing a function such as maximum, it's obvious that the mapper only has to output the maximum for each key it has seen. Doing so would reduce network traffic and increase performance. For a function such as average, it's a bit more complicated, but we can still redefine the algorithm such that for each mapper only one record is shuffled for each key.
- Using country from the patent data set as key illustrates data skew. The data is far from uniformly distributed, as a significant majority of the records would have U.S. as the key. Not only does every key/value pair in the input map to a key/value pair in the intermediate data, most of the intermediate key/value pairs will end up at a single reducer, overwhelming it.

Hadoop solves these bottlenecks by extending the MapReduce framework with a combiner step in between the mapper and reducer. You can think of the combiner as a helper for the reducer. It's supposed to whittle down the output of the mapper to lessen the load on the network and on the reducer. If we specify a combiner, the MapReduce framework may apply it zero, one, or more times to the intermediate data. In order for a combiner to work, it must be an equivalent transformation of the data with respect to the reducer. If we take out the combiner, the reducer's output will remain the same. Furthermore, the equivalent transformation property must hold when the combiner is applied to arbitrary subsets of the intermediate data.

If the reducer only performs a distributive function, such as maximum, minimum, and summation (counting), then we can use the reducer itself as the combiner. But many useful functions aren't distributive. We can rewrite some of them, such as averaging, to take advantage of a combiner.

The averaging approach taken by AverageByAttributeMapper.py is to output only each key/value pair. AverageByAttributeReducer.py will count the number of key/value pairs it receives and sum up their values, in order for a single final division to compute the average. The main obstacle to using a combiner is the counting operation, as the reducer assumes the number of key/value pairs it receives is the number of key/value pairs in the input data. We can refactor the MapReduce program to track the count explicitly. The combiner becomes a simple summation function with the distributive property.

Let's first refactor the mapper and reducer before writing the combiner, as the operation of the MapReduce job must be correct even without a combiner. We write the new averaging program in Java as the combiner must be a Java class.

NOTE The Streaming API allows you to specify a combiner using the `-combiner` option. Since Hadoop 0.21, the combiner can be either a script or a Java class. In addition, if you're using the `Aggregate` package, each value aggregator already has a built-in (Java) combiner. The `Aggregate` package will automatically use these combiners.

Let's write a Java mapper (listing 4.12) that's analogous to AverageByAttributeMapper.py of listing 4.7.

Listing 5.12 Java equivalent of AverageByAttributeMapper.py

```
public static class MapClass extends Mapper<LongWritable, Text, Text, Text> {
    public void map(LongWritable key, Text value,
                    Context context) throws
                    IOException, InterruptedException {

        String fields[] = value.toString().split(",", -20);
        String country = fields[4];
        String numClaims = fields[8];
        if (numClaims.length() > 0 && !numClaims.startsWith("\"")) {
            context.write(new Text(country),
                          new Text(numClaims + ",1"));
        }
    }
}
```

The crucial difference in this new Java mapper is that the output is now appended with a count of 1. We could've defined a new Writable data type that holds both the value and count, but things are simple enough that we're just keeping a comma-separated string in Text.

At the reducer, the list of values for each key are parsed. The total sum and count are then computed by summation and divided at the end to get the average.

```
public static class Reduce extends Reducer<Text, Text, Text, DoubleWritable> {

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

        double sum = 0;
        int count = 0;
        while (values.hasNext()) {
            String fields[] = values.next().toString().split(",");
            sum += Double.parseDouble(fields[0]);
            count += Integer.parseInt(fields[1]);
        }
        context.write(key, new DoubleWritable(sum/count));
    }
}
```

The logic of the refactored MapReduce job should not be too hard to follow. We added an explicit count for each key/value pair. This refactoring allows the intermediate data to be combined at each mapper before it's sent across the network.

Programmatically, the combiner must implement the `Reducer` interface. The combiner's `reduce()` method performs the combining operation. This may seem like a bad naming scheme, but recall that for the important class of distributive functions, the combiner and the reducer perform the same operations. Therefore, the combiner has adopted the reducer's signature to simplify its reuse. You don't have to rename your `Reduce` class to use it as a combiner class. In addition, because the combiner is performing an equivalent transformation, the type for the key/value pair in its output must match that of its input. In the end, we've created a `Combine` class that looks similar to the `Reduce` class, except it only outputs the (partial) sum and count at the end, whereas the reducer computes the final average.

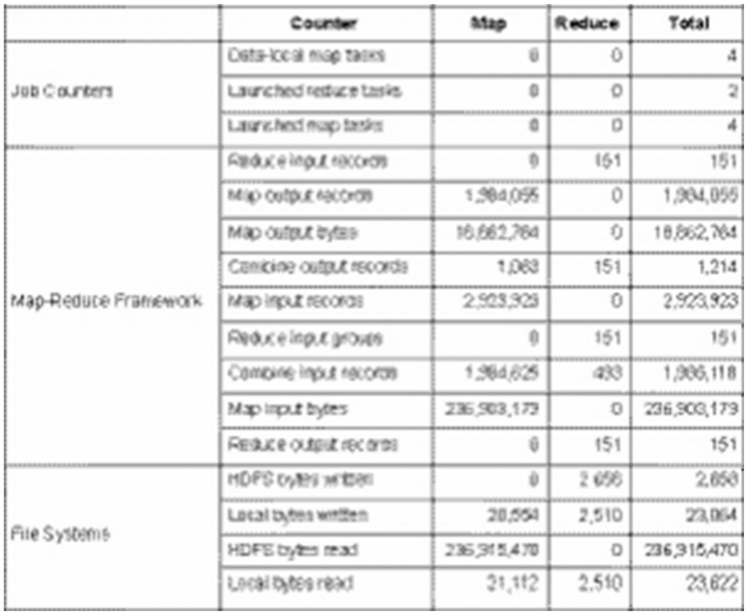
```
public static class Combine extends Reducer<Text, Text, Text, Text> {

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

        double sum = 0;
        int count = 0;
        while (values.hasNext()) {
            String fields[] = values.next().toString().split(",");
            sum += Double.parseDouble(fields[0]);
            count += Integer.parseInt(fields[1]);
        }
        context.write(key, new Text(sum + "," + count));
    }
}
```

To enable the combiner, the driver must specify the combiner's class to the `Job` object. You can do this through the `setCombinerClass()` method. The driver sets the mapper, combiner, and the reducer:

```
job.setMapperClass(MapClass.class);
job.setCombinerClass(Combine.class);
job.setReducerClass(Reduce.class);
```

A combiner doesn't necessarily improve performance. You should monitor the job's behavior to see if the number of records outputted by the combiner is meaningfully less than the number of records going in. The reduction must justify the extra execution time of running a combiner. You can easily check this through the JobTracker's Web UI, which we'll see in chapter 6.

Looking at figure 4.5, note that in the map phase, combine has 1,984,625 input records and only 1,063 output records. Clearly the combiner has reduced the amount of intermediate data. Note that the reduce side executes the combiner, though the benefit of this is negligible in this case.



	Counter	Map	Reduce	Total
Job Counters	Data-local map tasks	0	0	4
	Launched reduce tasks	0	0	2
	Launched map tasks	0	0	4
Map-Reduce Framework	Reduce input records	0	151	151
	Map output records	1,984,055	0	1,984,055
	Map output bytes	16,662,784	0	16,862,784
	Combine output records	1,063	151	1,214
	Map input records	2,923,923	0	2,923,923
	Reduce input groups	0	151	151
	Combine input records	1,984,625	493	1,985,118
	Map input bytes	236,903,179	0	236,903,179
	Reduce output records	0	151	151
File Systems	HDFS bytes written	0	2,656	2,656
	Local bytes written	20,554	2,510	23,064
	HDFS bytes read	236,315,470	0	236,315,470
	Local bytes read	21,112	2,510	23,622

Figure 4.5 Monitoring the effectiveness of the combiner in the `AveragingWithCombiner` job

5.6 Exercising what you've learned

Practice is the path to proficiency. You can try the following exercises to hone your ability to think in the MapReduce paradigm.

1. *Top K records*—Change AttributeMax.py (or AttributeMax.php) to output the entire record rather than only the maximum value. Rewrite it such that the MapReduce job outputs the records with the top `K` values rather than only the maximum.
2. *Web traffic measurement*—Take a web server log file and write a Streaming program with the `Aggregate` package to find the hourly traffic to that site.
3. *Inner product of two sparse vectors*—A vector is a list of values. Given two vectors, *X* = [*x1*, *x2*, ...] and *Y* = [*y1*, *y2*, ...], their inner product is *Z* = *x1* * *y1* + *x2* * *y2* + ... When *X* and *Y* are long but have many elements with zero value, they're usually given in a sparse representation:

   1,0.46

   9,0.21

   17,0.92

   ...

   where the key (first column) is the index into the vector. All elements not explicitly specified are considered to have a value of zero. Note that the keys don't need to be in a sorted order. In fact, the keys may not even be numerical. (For natural language processing, the keys can be words in a document, and the inner product is a measure of document similarity.) Write a `Streaming` job to compute the inner product of two sparse vectors. You can add a post-processing step after the MapReduce job to complete the computation.
4. *Time series processing*—Consider time-series data, where each record has a timestamp as key and a measurement (on that time period) as value. We want an output that is a linear function of the time series in a form: *y*(t) = *a*0 * *x*(t) + *a1* * *x*(t-1) + *a2* * *x*(t-2) + ... + *aN* * *x*(t-N) where t stands for time and *a0*,…,*aN* are known constants. In signal processing, this is known as an FIR filter. A particularly popular instance is the moving average, where *a0* = *a1* = … = *aN* = 1/N. Each point in *y* is the average of the previous N points in x. It's a simple way to smooth out time series.

   Implement this linear filter in MapReduce. Be sure to use a combiner. If you order the time series data chronologically (as they usually are) and N is relatively small, what's the reduction in network traffic for shuffling when a combiner is used? For extra credit, write your own partitioner so the output stays ordered chronologically.

   For the more advanced practitioners, this example illustrates the difference between scalability and performance. Implementing an FIR filter in Hadoop makes it scalable to process terabytes or more of data. Students of signal processing will recognize that a high performance implementation of an FIR filter often calls for a technique known as Fast Fourier Transform (FFT). A solution that is scalable and high performing would call for a MapReduce implementation of FFT, which is beyond the scope of this book.
5. *Commutative property*—Recall from basic math that the commutative property means the order of operation is irrelevant. For example, addition obeys the commutative

property, as a+b=b+a and a+b+c=b+a+c=b+c+a=c+a+b=c+b+a. Is the MapReduce framework fundamentally designed for implementing commutative functions? Why or why not?

6. *Multiplication (product)*—Many machine-learning and statistical-classification algorithms involve the multiplication of a large number of probability values. Usually we compare the product of one set of probabilities to the product of a different set, and choose a classification corresponding to the bigger product. We've seen that maximum is a distributive function. Is the product also distributive? Write a MapReduce program that multiplies all values in a data set. For full credit, apply the program to a reasonably large data set. Does implementing the program in MapReduce solve all scalability issues? What should you do to fix it? (Writing your own floating-point library is a popular answer, but not a good one.)
7. *Translation into fictional dialect*—A popular assignment in introductory computer science classes is to write a program that converts English to "pirate-speak." Many variations of the exercise exist for other semi-fictional dialects, such as "Snoop Dogg" and "E-40." Usually the solution involves a dictionary look-up for exact word matches ("for" becomes "fo," "sure" becomes "sho," "the" becomes "da," etc.), simple text rules (words ending in "ing" now ends in "in'," replace the last vowel of a word and everything after it with "izzle," etc.), and random injections ("kno' wha' im sayin'?"). Write such translations and use Hadoop to apply it to a large corpus such as Wikipedia.

5.7 Summary

MapReduce programs follow a template. Often the whole program is defined within a single Java class. Within the class, a driver sets up a MapReduce `Job` object, which is used as the blueprint for how the job is set up and run. You'll find the map and reduce functions in subclasses of `Mapper` and `Reducer`, respectively. Those classes are often no more than a couple dozen lines long, so they're usually written as inner classes for convenience.

Hadoop provides a Streaming API for writing MapReduce programs in a language other than Java. Many MapReduce programs are much easier to develop in a scripting language using the Streaming API, especially for ad hoc data analysis. The `Aggregate` package, when used with Streaming, enables one to rapidly write programs for counting and getting basic statistics.

MapReduce programs are largely about the map and the reduce functions, but Hadoop allows for a combiner function to improve performance by "pre-reducing" the intermediate data at the mapper before the reduce phase.

In standard programming (outside of the MapReduce paradigm), counting, summing, averaging, and so on are usually done through a simple, single pass of the data. Refactoring those programs to run in MapReduce, as we've done in this chapter, is relatively straightforward conceptually. More complex data analysis algorithms call for deeper reworking of the algorithms, which we cover in the next chapter.

5.8 Further resources

Although we've focused on the patent data sets in this chapter, there are other large publicly accessible data sets that you can download and play around with. Below are a few examples.

- http://www.netflixprize.com/index —Netflix is an online movie rental site. A crucial part of its business is a recommendation engine that suggests new movies to a user based on the user's ratings of previous movies. As part of a competition, it released a data set of user ratings to challenge people to develop better recommendation algorithms. The uncompressed data comes at 2 GB+. It contains 100 M+ movie ratings from 480 K users on 17 K movies. http://aws.amazon.com/publicdatasets/ —Amazon has hosted for free several large public data sets for its EC2 users. As of this writing, the data sets belong to the three categories of biology, chemistry, and economics. For example, one of the biological data sets is an annotated human genome data of roughly 550 GB. Under economics you can find data sets, such as the 2000 U.S. Census (approximately 200 GB).
- http://boston.lti.cs.cmu.edu/Data/clueweb09/—Carnegie Mellon University's Language Technologies Institute has released the ClueWeb09 data set to aid large-scale web research. It's a crawl of a billion web pages in 10 languages. The uncompressed data set takes up 25 TB. Given the size of the data set, the most efficient way to get it is in compressed form (which takes up 5 TB) shipped in hard disk drives. (At a certain scale, shipping hard drives through FedEx has a high "bandwidth.") As of this writing, CMU charges US$790 to ship four 1.5 TB drives with the compressed data.
- https://www.data.gov —the United States government publishes large amounts of data on a huge range of topics from agriculture to weather. Much of this data has already been aggregated, but much of it is also useful for data mash-ups with other data sources. For instance, census data per geographical region is available for marketing applications, and health spending statistics for medical informatics.

7 Programming practices

This chapter covers

- Best practices unique to developing Hadoop programs
- Debugging programs in local, pseudo-distributed, and fully distributed modes
- Sanity checking and regression testing program outputs
- Logging and monitoring
- Performance tuning

Now that you've gone through various programming techniques in MapReduce, this chapter will step back and cover programming practices.

Programming on Hadoop differs from traditional programming mainly in two ways. First, Hadoop programs are primarily about processing data. Second, Hadoop programs are run over a distributed set of computers. These two differences will change some aspects of your development and debugging processes, which we cover in sections 6.1 and 6.2.

Performance-tuning techniques tend to be specific to the programming platform, and Hadoop is no different. We cover tools and approaches to optimizing Hadoop programs in section 6.3.

Let's start with the development techniques applicable to Hadoop. Presumably you're already familiar with standard Java software engineering techniques. We focus on practices unique to data-centric programming within Hadoop.

7.1 Developing MapReduce programs

Chapter 2 discussed the three modes of Hadoop: local (standalone), pseudo-distributed, and fully distributed. They correspond roughly to development, staging, and production setups. Your development process will go through each of the three modes. You'll have to be able to switch between configurations easily. In practice you may even have more than one fully distributed cluster. Larger shops may, for example, have a “development” cluster to further harden MapReduce programs before running them on the real production cluster. You may have multiple clusters for different workloads. For example, there can be an in-house cluster for running many small to medium-sized jobs and a cluster in the cloud that’s more cost effective for running large, infrequent jobs.

While in previous versions of Hadoop, there was the hadoop-site.xml configuration file that centralized the different Hadoop configurations, with YARN the situation has become more complex. Hadoop is configured through a suite of files that break-up functionality for HDFS, the YARN daemons, and schedulers, making it much trickier to maintain multiple Hadoop environments that can easily be switched among during development and testing. Instead, it is easy to create several instances of Hadoop in different subdirectories and configure them to operate in local mode or as a single-node cluster for development purposes.

Before you run and test your Hadoop program, you’ll need to make data available for the configuration you’re running. Section 3.1 describes various ways to get data into and out of HDFS. For local and pseudo-distributed modes, you’ll only want a subset of your full data. Section 4.4 presents a Streaming program (RandomSample.py) that can randomly sample a percentage of records from a data set in HDFS. As it’s a Python script, you can also use it to sample down a local file with a Unix pipe:

```
cat datafile | RandomSample.py 10
```

will give you a 10 percent sample of the file datafile.

Now that you have all the different configurations set up and know how to put data into each configuration, let’s look at how to develop and debug in local and pseudo-distributed modes. The techniques build on top of each other as you get closer to the production environment. We defer the discussion of debugging on the fully distributed cluster ‘till the next section.

7.1.1 Local mode

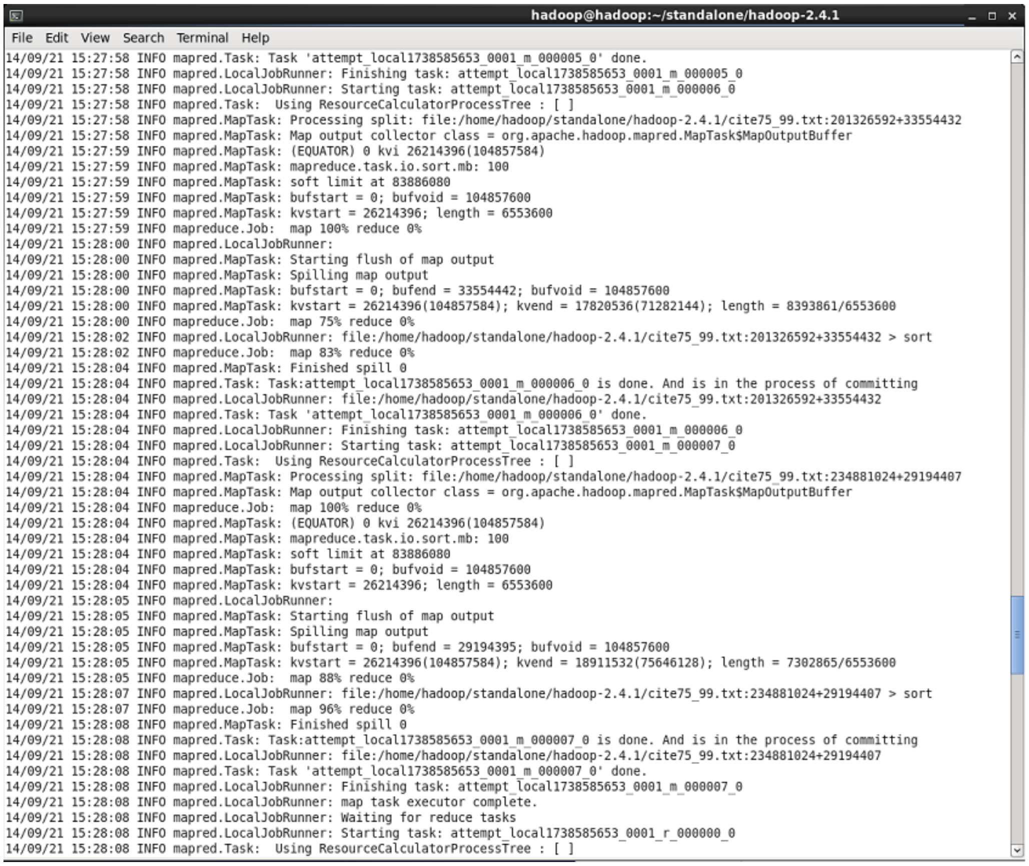
Hadoop in local mode runs everything within one single Java Virtual Machine (JVM) and uses the local filesystem (i.e., no HDFS). Running within one JVM allows you to use all the familiar Java development tools, such as a debugger. Using files from the local filesystem means you can quickly apply Unix commands or simple scripts on the input and output data. Examining files in HDFS, on the other hand, is limited to commands provided by the Hadoop command line. For example, to count how many records are in an output file, you can use `wc -l` if the file is in the local filesystem. If the file is in HDFS, then you’ll either have to write a MapReduce program or download the file to local storage before applying the Unix commands. As you’ll see, being able to access input and output files easily will be important to our development practices under local mode.

NOTE Local mode closely adheres to Hadoop’s MapReduce programming model, but it doesn’t support every feature. For example, it doesn’t support distributed cache, and it only allows a maximum of one reducer.

A program running in local mode will output all log and error messages to the console. It will also summarize the amount of data processed at the end. For example, running our skeleton MapReduce job (MyJob.java) to invert the patent citation data, the output is quite verbose, and Figure 7.1 is a snapshot in the middle of the job.

At the end of the job, Hadoop will print out the values of various internal counters. They're the number of records and bytes going through the different stages of MapReduce:

```
14/09/21 15:28:30 INFO mapred.Task: Task:attempt_local1738585653_0001_r_000000_0 is
      done. And is in the process of committing.

14/09/21 15:28:30 INFO output.FileOutputCommitter: Saved output of task
      'attempt_local1738585653_0001_r_000000_0' to
      file:/home/hadoop/standalone/hadoop-
      2.4.1/outcite/_temporary/0/task_local1738585653_0001_r_000000
```


hadoop@hadoop:~/standalone/hadoop-2.4.1
File Edit View Search Terminal Help

```
14/09/21 15:27:58 INFO mapred.Task: Task 'attempt_local1738585653_0001_m_000005_0' done.
14/09/21 15:27:58 INFO mapred.LocalJobRunner: Finishing task: attempt_local1738585653_0001_m_000005_0
14/09/21 15:27:58 INFO mapred.LocalJobRunner: Starting task: attempt_local1738585653_0001_m_000006_0
14/09/21 15:27:58 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ]
14/09/21 15:27:58 INFO mapred.MapTask: Processing split: file:/home/hadoop/standalone/hadoop-2.4.1/cite75_99.txt:201326592+33554432
14/09/21 15:27:58 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
14/09/21 15:27:59 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
14/09/21 15:27:59 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
14/09/21 15:27:59 INFO mapred.MapTask: soft limit at 83886080
14/09/21 15:27:59 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
14/09/21 15:27:59 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
14/09/21 15:27:59 INFO mapreduce.Job:  map 100% reduce 0%
14/09/21 15:28:00 INFO mapred.LocalJobRunner:
14/09/21 15:28:00 INFO mapred.MapTask: Starting flush of map output
14/09/21 15:28:00 INFO mapred.MapTask: Spilling map output
14/09/21 15:28:00 INFO mapred.MapTask: bufstart = 0; bufend = 33554442; bufvoid = 104857600
14/09/21 15:28:00 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 17820536(71282144); length = 8393861/6553600
14/09/21 15:28:00 INFO mapreduce.Job:  map 75% reduce 0%
14/09/21 15:28:02 INFO mapred.LocalJobRunner: file:/home/hadoop/standalone/hadoop-2.4.1/cite75_99.txt:201326592+33554432 > sort
14/09/21 15:28:02 INFO mapreduce.Job:  map 83% reduce 0%
14/09/21 15:28:04 INFO mapred.MapTask: Finished spill 0
14/09/21 15:28:04 INFO mapred.Task: Task:attempt_local1738585653_0001_m_000006_0 is done. And is in the process of committing
14/09/21 15:28:04 INFO mapred.LocalJobRunner: file:/home/hadoop/standalone/hadoop-2.4.1/cite75_99.txt:201326592+33554432
14/09/21 15:28:04 INFO mapred.Task: Task 'attempt_local1738585653_0001_m_000006_0' done.
14/09/21 15:28:04 INFO mapred.LocalJobRunner: Finishing task: attempt_local1738585653_0001_m_000006_0
14/09/21 15:28:04 INFO mapred.LocalJobRunner: Starting task: attempt_local1738585653_0001_m_000007_0
14/09/21 15:28:04 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ]
14/09/21 15:28:04 INFO mapred.MapTask: Processing split: file:/home/hadoop/standalone/hadoop-2.4.1/cite75_99.txt:234881024+29194407
14/09/21 15:28:04 INFO mapred.MapTask: Map output collector class = org.apache.hadoop.mapred.MapTask$MapOutputBuffer
14/09/21 15:28:04 INFO mapreduce.Job:  map 100% reduce 0%
14/09/21 15:28:04 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
14/09/21 15:28:04 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
14/09/21 15:28:04 INFO mapred.MapTask: soft limit at 83886080
14/09/21 15:28:04 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
14/09/21 15:28:04 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
14/09/21 15:28:05 INFO mapred.LocalJobRunner:
14/09/21 15:28:05 INFO mapred.MapTask: Starting flush of map output
14/09/21 15:28:05 INFO mapred.MapTask: Spilling map output
14/09/21 15:28:05 INFO mapred.MapTask: bufstart = 0; bufend = 29194395; bufvoid = 104857600
14/09/21 15:28:05 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend = 18911532(75646128); length = 7302865/6553600
14/09/21 15:28:05 INFO mapreduce.Job:  map 88% reduce 0%
14/09/21 15:28:07 INFO mapred.LocalJobRunner: file:/home/hadoop/standalone/hadoop-2.4.1/cite75_99.txt:234881024+29194407 > sort
14/09/21 15:28:07 INFO mapreduce.Job:  map 96% reduce 0%
14/09/21 15:28:08 INFO mapred.MapTask: Finished spill 0
14/09/21 15:28:08 INFO mapred.Task: Task:attempt_local1738585653_0001_m_000007_0 is done. And is in the process of committing
14/09/21 15:28:08 INFO mapred.LocalJobRunner: file:/home/hadoop/standalone/hadoop-2.4.1/cite75_99.txt:234881024+29194407
14/09/21 15:28:08 INFO mapred.Task: Task 'attempt_local1738585653_0001_m_000007_0' done.
14/09/21 15:28:08 INFO mapred.LocalJobRunner: Finishing task: attempt_local1738585653_0001_m_000007_0
14/09/21 15:28:08 INFO mapred.LocalJobRunner: map task executor complete.
14/09/21 15:28:08 INFO mapred.LocalJobRunner: Waiting for reduce tasks
14/09/21 15:28:08 INFO mapred.LocalJobRunner: Starting task: attempt_local1738585653_0001_r_000000_0
14/09/21 15:28:08 INFO mapred.Task:  Using ResourceCalculatorProcessTree : [ ]
```

Figure 7.1 Running a Hadoop program in local mode outputs all the log messages to the console.

```
14/09/21 15:28:30 INFO mapred.LocalJobRunner: Finishing task:
      attempt_local1738585653_0001_r_000000_0 14/09/21 15:28:30 INFO
      mapred.LocalJobRunner: reduce task executor complete. 14/09/21 15:28:30 INFO
      mapreduce.Job:  map 100% reduce 100% 14/09/21 15:28:30 INFO mapreduce.Job: Job
      job_local1738585653_0001 completed successfully  1
14/09/21 15:28:30 INFO mapreduce.Job: Counters: 33
      File System Counters
        FILE: Number of bytes read=2062128089
        FILE: Number of bytes written=2216635231
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
      Map-Reduce Framework
        Map input records=16522439
        Map output records=16522439
        Map output bytes=264075431
        Map output materialized bytes=297120357
        Input split bytes=960
        Combine input records=0
        Combine output records=0
        Reduce input groups=3258984
        Reduce shuffle bytes=297120357
        Reduce input records=16522439
        Reduce output records=16522439
        Spilled Records=33044878
        Shuffled Maps =8
        Failed Shuffles=0
        Merged Map outputs=8
        GC time elapsed (ms)=562
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=1849290752
      Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
      File Input Format Counters
        Bytes Read=264104103
      File Output Format Counters
        Bytes Written=266138531
```

The input and output of the MapReduce job are both in the local filesystem. We can examine them using standard Unix commands such as `wc -l` or `head`. As we are deliberately using smaller data sets during development, we can even load them into a text editor or a spreadsheet. We can use the many features of those applications to sanity check the correctness of our program.

SANITY CHECKING

Most MapReduce programs involve at least some counting or arithmetic, and bugs (especially typos) in mathematical programming don't call attention to themselves in the form of thrown exceptions or threatening error messages. Your math can be wrong even though your program is technically "correct," and everything will run smoothly, but the end result will be useless. There's no simple way to uncover arithmetic mistakes, but some sanity checking will go a long way. At a high level you can look at the overall count, maximum, average, and so on, of various metrics and see if they match expectation. At a low level you can pick a particular output record and verify that it was produced correctly. For example, when we created the inverted citation graph, the first few lines were

```
"CITED" "CITING"
1       3964859,4647229
10000   4539112
100000 5031388
1000006 4714284
1000007 4766693
1000011 5033339
1000017 3908629
1000026 4043055
1000033 4190903,4975983
```

Our job concludes that patent number 1 is cited twice, by 3964859 and 4647229. We can verify this claim by grepping over the sampled input data to look for records where patent number 1 is cited.

```
grep ",1$" input/cite75_99.txt
```

We indeed get the two records as expected. You can verify a few more records to gain confidence in the correctness of your program's math and logic.[15]

An eyesore about the output of this inverted citation graph is that the first line is not real data.

```
"CITED" "CITING"
```

It's an artifact from the first line of the input data being used as data definition. Let's add some code to our mapper to filter out non-numeric keys and values, and in the process demonstrate regression testing.

REGRESSION TESTING

Our data-centric approach to regression testing revolves around "diff'ing" various output files from before and after code changes. For our particular change, we should only be taking out one line from the job's output. To verify that this indeed is the case, let's first save the output of our current job. In local mode, we have a maximum of only one reducer, so the job's output is only one file, which we call job_1_output.

For regression testing, it's also useful to save the output of the map phase. This will help us isolate bugs to either the map phase or the reduce phase. We can save the output of the

[15] In this case, you may suspect whether patent number 1 is really cited by those two patents. The number 1 feels wrong, an outlier in the range of patent numbers being cited. There can be mistakes in the original input data. We have to track down the patents themselves if we want to verify this. In any case, ensuring data quality is an important topic but is beyond our discussion of Hadoop.

map phase by running the MapReduce job with zero reducers. We can do this easily using the `-Dmapred.reduce.tasks=0` option. In this mapper-only job, there will be multiple files as each map task will write its output to its own file. Let's copy all of them into a directory called job_1_intermediate.

Having stored away the output files, we can make the desired code changes to the `map()` method in `MapClass`. The code itself is trivial. We focus on testing it.

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

    try
    {
        if (Integer.parseInt(key.toString()) > 0 &&
            Integer.parseInt(value.toString()) > 0)
        {
            context.write(value, key);
        }
    } catch (NumberFormatException e) { }
}
```

Compile and execute the new code against the same input data. Let's run it as a map-only job first and compare the intermediate data. As we've only changed the mapper, any bug should first manifest in differences in the intermediate data.

```
diff output/job_1_intermediate/ output/test/
```

We get the following output from the diff utility:

```
Binary files output/job_1_intermediate/.part-00000.crc and
➥ output/test/.part-00000.crc differ
diff output/job_1_intermediate/part-00000 output/test/part-00000
1d0
< "CITED"      "CITING"
```

We found differences in the binary file .part-00000.crc. This is an internal file for HDFS to keep checksums for the file part-00000. A difference in checksum means that part-00000 has changed, and diff prints out the exact differences later. The new intermediate file, under output/test, is missing the quoted field descriptors. More importantly, we find no other changes. So far so good. If we run the whole job with one reducer, we expect the final output to differ by one line too.

Well, it turns out not to be the case. If you run the whole job with one reducer and compare the final output with job_1_output from the original run, you'll find many differences. What do you think happened? Let's look at the first few lines of the diff to find out.

```
$ diff output/job_1_output output/test/part-00000 | head -n 15
1,2c1
< "CITED"      "CITING"
< 1     3964859,4647229
---
> 1     4647229,3964859
19c18
< 1000067      5312208,4944640,5071294
```

```
---
> 1000067       4944640,5071294,5312208
22,23c21,22
< 1000076       4867716,5845593
< 1000083       5566726,5322091
---
> 1000076       5845593,4867716
> 1000083       5322091,5566726
```

We see that the line with the field descriptors (“CITED” and “CITING”) are taken out as expected. As to the rest of the differences, there’s a definite pattern.

In our `reduce()` method, we have concatenated the list of values for each key in the order Hadoop has given them to us. Hadoop doesn’t provide any guarantee as to the order of those values. We see that taking out one line in the intermediate data impacts the order of the values for many keys at the reducer. As we know that this job’s correctness is invariant to the ordering, we can ignore the differences. Regression testing is inherently conservative and tends to set off false alarms. You should use it with that in mind.

We have advocated the use of a sampled data set for development, because it is more representative of the structure and properties of the data set we use in production. We have used the same sampled data set for regression testing, but you can also manually construct a separate input data set with edge cases that are atypical of the production data. For example, you may put in empty values or extra tab characters or other unusual records in this constructed data set. This test data set is for ensuring that your program continues to handle the edge cases even as it evolves. This test data set doesn’t need to have more than several dozen records. You can visually inspect the entire output to see if your program still functions as expected.

CONSIDER USING LONG INSTEAD OF INT

Most Java programmers instinctively default to the `int` type (or `Integer` or `IntWritable`) to represent integer values. In Java the `int` type can hold any integer between 231-1 and -231, or between 2,147,483,647 and -2,147,483,648. This is adequate for most applications. Rarely do programmers put too much thought into it. When you’re processing Hadoop-scale data, it’s not unusual for some counter variables to need a bigger range. You won’t see this requirement under your development data set, which by design is small. It may not even matter to your current production data, but as your business operation grows, your data set will get bigger. It may get to a point where some variables will outgrow the `int` range and cause arithmetic errors. Take the canonical word counting example. When processing millions of documents, you won’t have any word count that goes beyond 2 billion,[16] and an `int` is adequate. But as you grow to process tens of millions or hundreds of millions of documents, counting frequent words like the can cross the limit of an `int` type. Rather than wait for this kind of bug to creep up on you in production, which is much harder to debug and costlier to fix, now is the time

[16] This is not absolutely true and will depend on your documents’ size and content.

you should go through your code and carefully consider whether your numeric variables should be long or `LongWritable` to handle future scale.[17]

7.1.2 Pseudo-distributed or Single Node Cluster mode

Local mode has none of the distributed characteristics of a production Hadoop cluster. We may not see many bugs when running in local mode. Hadoop provides a pseudo-distributed mode that has all the functionality and "nodes" of a production cluster—NameNode, SecondaryNameNode, DataNode, and YARN daemons, each running on a separate JVM. Pseudo-distributed mode is now referred to as a Single Node Cluster at the Apache Hadoop site. All the software components are distributed, and pseudo-distributed mode differs from a production cluster only at the system and hardware level. It uses only one physical machine—your own local computer. We should make sure our jobs can run in pseudo-distributed mode before deploying them to a full production cluster.

Chapter 2 describes the configuration and commands to start pseudo-distributed mode. You'll start all the daemons on your computer to make it function like a cluster. You interface with it as if it is a distinct Hadoop cluster. You put data into its own HDFS filesystem. You submit your jobs to it for running rather than run them in the same user space. Most importantly, you now monitor it "remotely" through log files and the web interface. You'll use the same tools later to monitor a production cluster.

LOGGING

Let's run in the pseudo-distributed cluster the same job we had in local mode. You put the input file into HDFS using the `hadoop fs` command. Submit the job for running using the same `hadoop jar` command as in local mode.

In previous versions of Hadoop, the console in pseudo-distributed mode produced less feedback and fewer counters than in local mode. With Hadoop 2 the range of counters and information increased again. Like before, Hadoop hasn't stopped outputting debugging messages. In fact, it's outputting much more now. These messages are also saved into log files.

You can find the log files under the /logs directory. Different services (NameNode, YARN ResourceManager, etc.) create separate log files. The filename distinguish the service logging a file. Hadoop rotates log files daily. The most recent one ends in .log. It further appends the older ones with their date. Under the default setting, Hadoop doesn't delete old log files automatically. You should proactively archive and delete them to make sure they're not taking up too much space.

[17] The problem of exceeding a numeric range is not unique to Hadoop. You'll remember the famous Y2K problem where older programs only allocated two digits to represent year. More recently, almost all web operations that have experienced explosive growth (such as Facebook, Twitter, and RockYou) have had to retool their systems to handle a bigger range of user IDs or document IDs than they originally expected.

Log files for the NameNode, SecondaryNameNode, DataNode, and YARN daemons are used for debugging the respective services. They're not too important in pseudo-distributed mode. In production clusters, you as a system administrator can look at them to debug problems in those corresponding nodes.

Your MapReduce program can output to STDOUT and STDERR (System.out and System.err in Java) its own logging messages. Hadoop records those under files named stdout and stderr, respectively. There will be a distinct file for each task attempt. (A task can have more than one attempt if the first one fails.) These user log files are under the /logs/userlogs subdirectory.

Besides logging to STDOUT and STDERR, your program can also send out live status messages using the `setStatus()` method on the Context object being passed to the `map()` and `reduce()` methods. (For Streaming programs, the status information is updated by sending a string of the form *reporter*:*status*:*message* to STDERR.) This is useful for long-running jobs where you can monitor them as they run. The status message is shown on the ApplicationMaster web UI, which can be accessed from the ResourceManager UI, to be described next.

RESOURCEMANAGER WEB UI

By definition events occur in many different places in a distributed program. This makes monitoring more difficult. The system becomes more like a black box, and we need specialized monitoring tools to peek into the various states within it. The YARN ResourceManager provides a web interface for tracking the progress and various states of your jobs. Under the default configuration, you can set your browser to

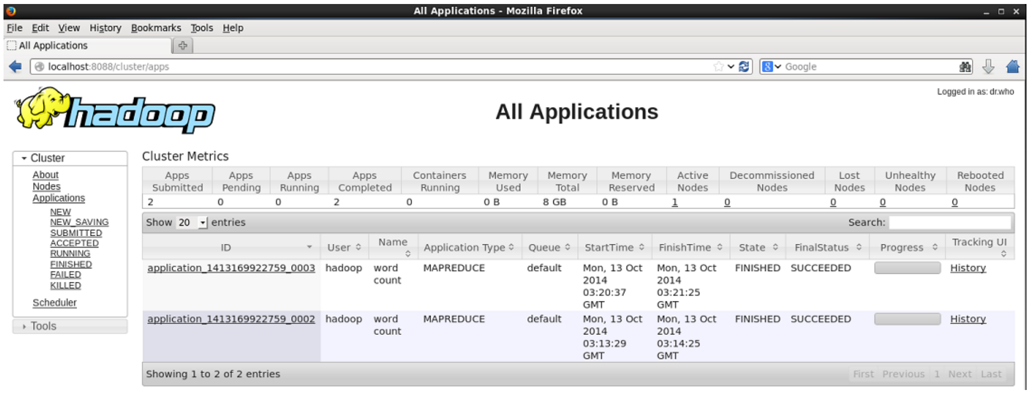
```
http://localhost:8088/cluster
```

to view the starting page of the administration tool for your pseudo-distributed cluster.[18] It shows a summary of the Hadoop cluster, as well as lists of jobs that are running, completed, and failed. See Figure 7.3.

Hadoop tracks jobs internally by their job ID. A job ID is a string prefixed with *job_*, followed by the cluster ID (which is a timestamp of when the cluster was started), followed by an auto-incremented job number. In Hadoop 2, YARN creates an application that executed the MapReduce jobs, so the ResourceManager shows the application ID. While the application is running, the individual jobs can be tracked as well through the ApplicationMaster UI. The web UI lists each job with the user name and application name. In pseudo-distributed mode, it's relatively easy to identify the application you're currently working with, as you'll run one application and hence job at a time. When you get to a multiuser production environment, you'll have to narrow down your applications by looking for your Hadoop user name and the name of your current application. For MapReduce applications, the name of your application is

[18] In fully distributed mode, replace "localhost" with the domain of the JobTracker master.

just the job name set through the job object using the `setJobName()` method of the `Job` object. The name for a Streaming job is set through a configuration property shown in Table 7.1.



All Applications - Mozilla Firefox

File Edit View History Bookmarks Tools Help

All Applications

localhost:8088/cluster/apps

Logged in as: dr.who

All Applications

Cluster
About
Nodes
Applications
NEW
NEW_SAVING
SUBMITTED
ACCEPTED
RUNNING
FINISHED
FAILED
KILLED
Scheduler
Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
2	0	0	2	0	0 B	8 GB	0 B	1	0	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1413169922759_0003	hadoop	word count	MAPREDUCE	default	Mon, 13 Oct 2014 03:20:37 GMT	Mon, 13 Oct 2014 03:21:25 GMT	FINISHED	SUCCEEDED		History
application_1413169922759_0002	hadoop	word count	MAPREDUCE	default	Mon, 13 Oct 2014 03:13:29 GMT	Mon, 13 Oct 2014 03:14:25 GMT	FINISHED	SUCCEEDED		History

Showing 1 to 2 of 2 entries First Previous 1 Next Last

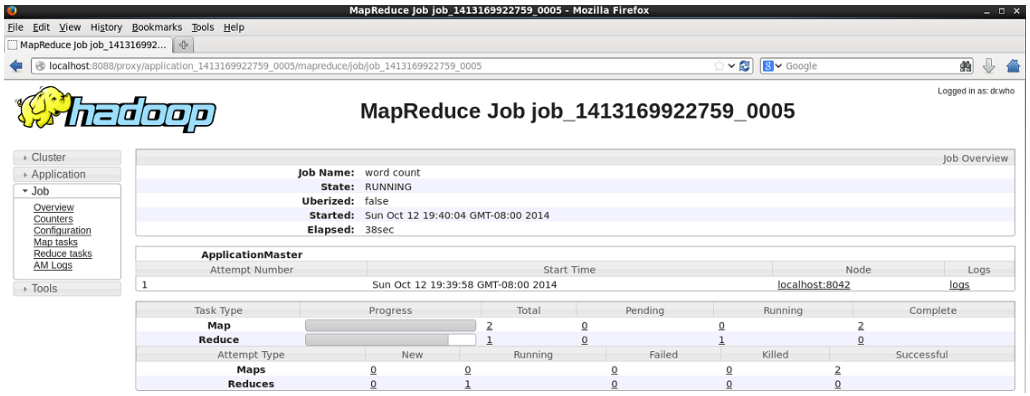
Figure 7.3 The ResourceManager web UI main page

Table 7.1 Configuration property for setting a job's name

Property	Description
`mapreduce.job.name`	String property denoting the name of a job

On the administration page, you can see each application with the total Progress for the application. While running, the ApplicationMaster UI shows the completion percentage of its map phase. It shows the number of map tasks for the job and the number of completed ones. You can see the same metrics for the reduce side. This gives you a rough summary of your job's progress. To drill down more on a particular job, you can click on the job ID, which is a link that'll take you to the job's administration page. See Figure 7.4.

The job page shows the volume of various input/output due to the running of the job. The page refreshes itself periodically but you can also refresh the page manually to get the updated numbers. You can start exploring the various aspects of your job from the many links on this page. For example, clicking on the map link will take you to a list of all map tasks for the job. See Figure 7.5.



MapReduce Job job_1413169922759_0005 - Mozilla Firefox

localhost:8088/proxy/application_1413169922759_0005/mapreduce/job/job_1413169922759_0005

Logged in as: dr.who

MapReduce Job job_1413169922759_0005

Cluster
Application
Job
Overview
Counters
Configuration
Map tasks
Reduce tasks
AM Logs
Tools

Job Overview

Job Name:	word count
State:	RUNNING
Uberized:	false
Started:	Sun Oct 12 19:40:04 GMT-08:00 2014
Elapsed:	38sec

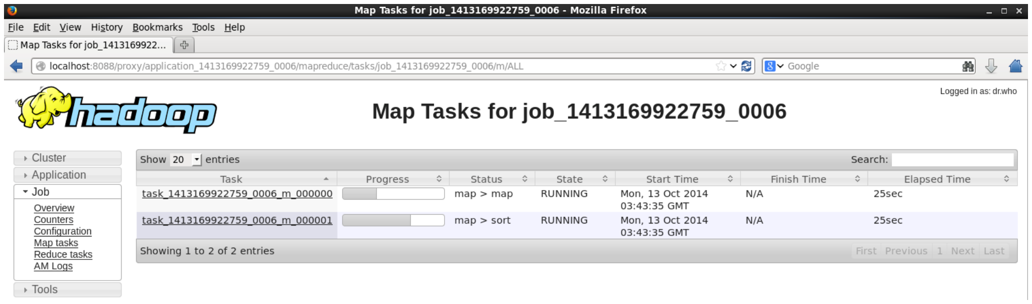
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Sun Oct 12 19:39:58 GMT-08:00 2014	localhost:8042	logs

Task Type	Progress	Total	Pending	Running	Complete
Map		2	0	0	2
Reduce		1	0	1	0

Attempt Type	New	Running	Failed	Killed	Successful
Maps	0	0	0	0	2
Reduces	0	1	0	0	0

Figure 7.4 The ApplicationMaster's administration page for a single job



Map Tasks for job_1413169922759_0006 - Mozilla Firefox

localhost:8088/proxy/application_1413169922759_0006/mapreduce/tasks/job_1413169922759_0006/m/ALL

Logged in as: dr.who

Map Tasks for job_1413169922759_0006

Cluster
Application
Job
Overview
Counters
Configuration
Map tasks
Reduce tasks
AM Logs
Tools

Show 20 entries   Search:

Task	Progress	Status	State	Start Time	Finish Time	Elapsed Time
task_1413169922759_0006_m_000000		map > map	RUNNING	Mon, 13 Oct 2014 03:43:35 GMT	N/A	25sec
task_1413169922759_0006_m_000001		map > sort	RUNNING	Mon, 13 Oct 2014 03:43:35 GMT	N/A	25sec

Showing 1 to 2 of 2 entries   First Previous 1 Next Last

Figure 7.5 List of tasks in the web UI. This figure shows all the map tasks for a single job. Each task can update its own status message.

Tasks are identified by a task ID. To construct the task ID, you start with the job ID the task runs under but replace the job_ prefix with task_. You then append it with _m for a map task or _r for a reduce task. You further append it with an auto-incremented number within each group. In the ApplicationMaster web UI, you'll see each task with its status, which you can programmatically set through the `setStatus()` method described earlier.

Clicking a task ID will bring you to a page that further describes different attempts of a task. Hadoop makes several retry attempts at a failed task before failing the entire job. The different job and application panels in the ResourceManager and ApplicationMaster UIs provide many other links and metrics. Most should be self-explanatory.

KILLING JOBS

Unfortunately, sometimes a job goes awry after you've started it but it doesn't actually fail. It may take a long time to run or may even be stuck in an infinite loop. In (pseudo-) distributed mode you can manually kill a job using the command

```
bin/hadoop job -kill job_id
```

where *job_id* is the job's ID as given in JobTracker's web UI.

7.2 Monitoring and debugging on a production cluster

After successfully running your job in a pseudo-distributed cluster, you're ready to run it on a production cluster using real data. We can apply all the techniques we've used for development and debugging on the production cluster, although the exact usage may be slightly different. Your cluster should still have a ResourceManager Web UI, but the domain is no longer localhost. It's now the address of the cluster's ResourceManager. The port number will still be 8088 unless it's been configured differently.

In pseudo-distributed mode, when there's only one node, all the log files are in a single /logs directory that you can access locally. In a fully distributed cluster, each node has its own /logs directory to keep its log files. Getting access to those log files required logging into the different datanodes prior to Hadoop 2. With YARN, however, a new service is available to manage access to job histories: the MR Jobhistory Daemon. This can be run in pseudo-distributed mode but it really becomes useful in full clusters.

Starting and stopping the MR Jobhistory daemon is done using a script in the sbin subdirectory:

```
sbin/mr-jobhistory-daemon.sh start historyserver
```

The jobhistory daemon aggregates together all of the logs for a job and provides a convenient web portal to access them. On a pseudo-distributed set-up, the web UI is found at this location:

```
http://localhost:19888/jobhistory
```

In addition to the development and testing techniques we've mentioned so far, you also have monitoring and debugging techniques that are more useful in a production cluster on real data, which we explore in this section.

7.2.1 Counters

You can instrument your Hadoop job with counters to profile its overall operation. Your program defines various counters and increments their counts in response to specific events. Hadoop automatically sums the same counter from all tasks (of the same job) so that it reflects the profile of the overall job. It displays the value of your counters in the MR Jobhistory Daemon web UI along with Hadoop's internal counters.

The canonical application of counters is for tracking different input record types, particularly for tracking "bad" records. Recall from section 4.4 our example for finding the average number of claims for patents from each country. We know the number of claims is not available for many records. Our program skips over those records, and it's useful to know the number of records we're skipping. Beyond satisfying our curiosity, such instrumentation allows us to understand the program's operation and do some "reality checks" for its correctness.

We use counters by first getting them through the `Context.getCounter()` method, passing in a group name and a counter name. The Context object is passed to the `map()` and reduce() methods. Once you have the counter, you call increment() with the amount to increment. You use uniquely named counters for each different event. When you call getCounter() with a new counter group and name, that counter is initialized.

The Context.getCounter() method has two signatures, depending on how you want to specify a counter's name:

```
Counter getCounter(String group, String counter)
Counter getCounter(Enum<?> name)
```

The first form is more general in that it allows you to specify the counter name with dynamic strings at run time. The combination of two strings, group and counter, uniquely defines a counter. When counters are reported (in the Web UI or as text at the end of a job run), counters of the same group are reported together.

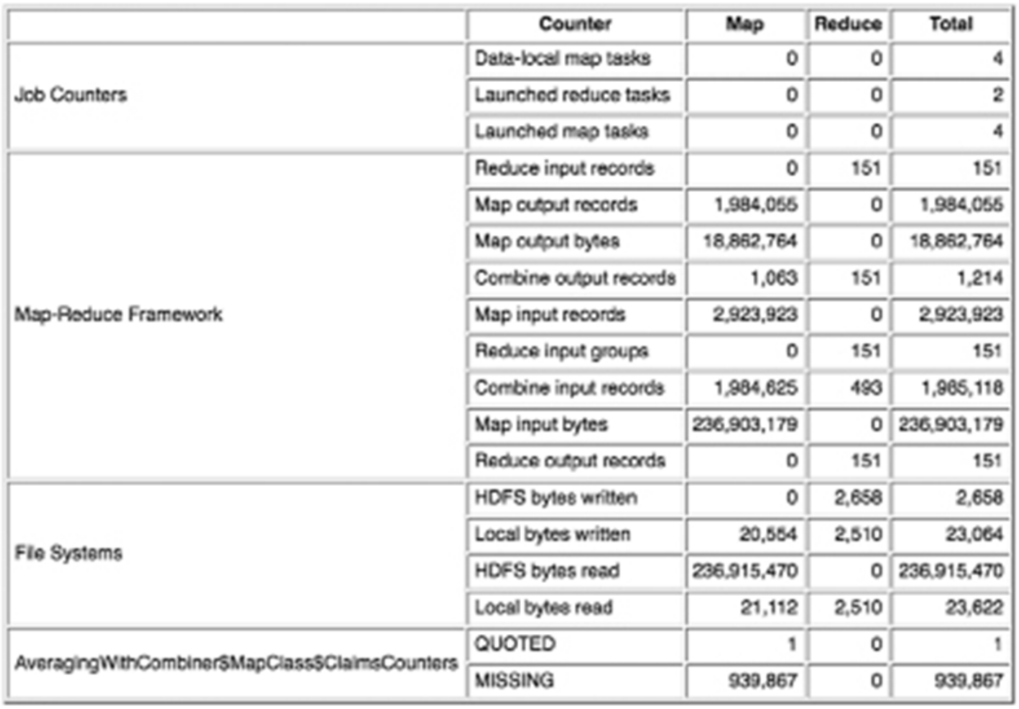
The second form uses a Java enum to specify counter names, which forces you to have them defined at compile time, but it also allows for type checking. The enum's name is used as the group string, whereas the enum's field is used as the counter string.

Listing 7.1 is the MapClass from listing 4.12 rewritten with counters to track the number of missing values and "quoted" values. (Only the first row of column description should be a "quoted" value.) An enum called ClaimsCounters is defined with values MISSING and QUOTED. Logic in the code increments the counters to reflect the record it's processing.

Listing 7.1 A MapClass with Counters to count the number of missing values

```
public static class MapClass extends Mapper<LongWritable, Text, Text, Text> {

    static enum ClaimsCounters { MISSING, QUOTED };

    public void map(LongWritable key, Text value,
                    Context context) throws IOException {

        String fields[] = value.toString().split(",", -20);
        String country = fields[4];
        String numClaims = fields[8];

        if (numClaims.length() == 0) {
            context.getCounter(ClaimsCounters.MISSING).increment(1);
        } else if (numClaims.startsWith("\"")) {
            context.getCounter(ClaimsCounters.QUOTED).increment(1);
        } else {
            context.write(new Text(country), new Text(numClaims + ",1"));
        }
    }
}
```

After running the program, we can see the defined counters along with Hadoop's internal counters in the Jobhistory web UI. See Figure 7.6.



	Counter	Map	Reduce	Total
Job Counters	Data-local map tasks	0	0	4
	Launched reduce tasks	0	0	2
	Launched map tasks	0	0	4
Map-Reduce Framework	Reduce input records	0	151	151
	Map output records	1,984,055	0	1,984,055
	Map output bytes	18,862,764	0	18,862,764
	Combine output records	1,063	151	1,214
	Map input records	2,923,923	0	2,923,923
	Reduce input groups	0	151	151
	Combine input records	1,984,625	493	1,985,118
	Map input bytes	236,903,179	0	236,903,179
	Reduce output records	0	151	151
File Systems	HDFS bytes written	0	2,658	2,658
	Local bytes written	20,554	2,510	23,064
	HDFS bytes read	236,915,470	0	236,915,470
	Local bytes read	21,112	2,510	23,622
AveragingWithCombiner$MapClass$ClaimsCounters	QUOTED	1	0	1
	MISSING	939,867	0	939,867

Figure 7.6 Jobhistory web UI collects and shows the counter information.

We see that the `enum`'s fully qualified Java name (with `$` to separate out the inner class hierarchy) is used as the group name. The fields MISSING and QUOTED are used to define separate counters. As expected, it increments the QUOTED counter only once and the MISSING counter 939,867 times. Does the data set have that many rows with missing claim counts? The originator of the data set stated that claim counts are missing for patents granted before 1975. Merely eyeballing figure 4.3, we guess that about a third of all the patents in our data set are granted before 1975. Looking at the map input records count (from Figure 7.6) we see there's a total of 2.9M+ records. The numbers seem consistent and we can feel more confident about the correctness of the processing.

A Streaming process can also use counters. It needs to send a specially formatted line to STDERR in the form of

```
reporter:counter:group,counter,amount
```

where *group*, *counter*, and *amount* are the corresponding arguments one would've passed to `getCounter()` and `increment()` in Java. For example, in Python one can increment the `ClaimsCounters.MISSING` counter with

```
sys.stderr.write("reporter:counter:ClaimsCounters,MISSING,1\n")
```

Be sure to include the newline character (`\n`) at the end. Hadoop Streaming will not properly interpret the string without that.

7.2.2 Skipping bad records

When dealing with large data sets, it is inevitable that some records will have errors. It's not unusual to focus several iterations of your development cycle on making the program robust to unexpected data.[19] Your program may never be completely foolproof, though. Your program will process new data, and new data will think of new ways to misbehave. You may even be using a parser that depends on third-party libraries you have no control over. While you should make your program as robust as possible to malformed records, you should also have a recovery mechanism to handle the cases you couldn't plan for. You don't want your whole job to fail only because it fails to handle one bad record.

Hadoop's mechanism for recovering from hardware failures doesn't work for recovering from deterministic software failures caused by bad records. Instead it provides a feature for skipping over records that it believes to be crashing a task. If this skipping feature is on, a task will enter into skipping mode after the task has been retried several times. Once in skipping mode, the system will track and determine which record range is causing failure. The ApplicationMaster will then restart the task but skip over the bad record range.

CONFIGURING RECORD SKIPPING IN JAVA

The skipping feature was available as early as Hadoop version 0.19, but it's disabled by default. In Java, the feature is controlled through the class `SkipBadRecords`, which consists entirely of static methods. The job driver needs to call one or both methods:

```
public static void setMapperMaxSkipRecords(Configuration conf,
        long maxSkipRecs)
public static void setReducerMaxSkipGroups(Configuration conf,
        long maxSkipGrps)
```

to turn on record skipping for map tasks and reduce tasks, respectively. The driver calls the methods with the configuration object and the maximum number of records in a skip range. If the maximum skip range size is set to 0 (default), then record skipping is disabled. Hadoop finds the skip range using a divide-and-conquer approach. It executes the task with the skip range halved each time, and determines the half with the bad record(s). The process iterates until the skip range is within the acceptable size. This is a rather expensive operation, particularly if the maximum skip range size is small. You may need to increase the maximum number of task attempts in Hadoop's normal task recovery mechanism to accommodate the extra attempts. You can do this using the methods `Job.setMaxMapAttempts()` and

[19] Unexpected data are not always mistakes. Someone once told me he had a program that was crashing in processing users' geographical information. Further digging revealed that one user was from a real city named Null.

Job.setMaxReduceAttempts(), or set the equivalent properties mapreduce.map.maxattempts and mapreduce.reduce.maxattempts.

If skipping is enabled, Hadoop enters skipping mode after the task has failed twice. You can set the number of task failures needed to trigger skipping mode in SkipBadRecords' setAttemptsToStartSkipping() method:

```
public static void setAttemptsToStartSkipping(Configuration conf,
        int attemptsToStartSkipping)
```

Hadoop will log skipped records to HDFS for later analysis. They're written as sequence files in the _log/skip directory. We cover sequence files in more detail in section 6.3.3. For now you can think of it as a Hadoop-specific compressed format. It can be uncompressed and read using the command:

```
bin/hadoop fs -text <filepath>
```

You can change the log directory for skipped records from _log/skip using the method SkipBadRecords.setSkipOutputPath(JobConf conf, Path path). If path is set to null or to a Path with a string value of "none", Hadoop will not record the skipped records.

CONFIGURING RECORD SKIPPING OUTSIDE OF JAVA

Although you can set the record-skipping feature in Java by calling methods in SkipBadRecords in your driver, sometimes you may want to set this feature using the generic options available in GenericOptionsParser instead. This is because the person running the program can have a better idea about the range of bad records to expect and set the parameters more appropriately than the original developer. Furthermore, Streaming programs can't access SkipBadRecords; the record skipping features must be configured using Streaming's -D property (-jobconf in version 0.18). Table 7.2 shows the Job properties being set by the SkipBadRecords method calls.

Table 7.2 Equivalent JobConf properties to method calls in SkipBadRecords

SkipBadRecords method	JobConf property
setAttemptsToStartSkipping()	mapreduce.task.skip.start.attempts
setMapperMaxSkipRecords()	mapreduce.map.skip.maxrecords
setReducerMaxSkipGroups()	mapreduce.reduce.skip.maxgroups
setSkipOutputPath()	mapreduce.job.skip.outdir
setAutoIncrMapperProcCount()	mapreduce.map.skip.proc-count.auto-incr
setAutoIncrReducerProcCount()	mapreduce.reduce.skip.proc-count.auto-incr

We haven't explained the last two properties yet. Their default values are fine for most Java programs but we need to change them for Streaming ones.

In determining the record range to skip, Hadoop needs an accurate count of the number of records a task has processed. Hadoop uses an internal counter and by default it's incremented after each call to the map (reduce) function. For Java programs this is a good approach to track the number of records processed. It can break down in some cases, such as programs that process records asynchronously (say, by spawning threads) or buffer them to process in chunks, but it usually works. In Streaming programs, this default behavior wouldn't work at all because there's no equivalent of the map (reduce) function that gets called to process each record. In those situations you have to disable the default behavior by setting the Boolean properties to false, and your task has to update the record counters itself.

In Python, the map task can update the counter with

```
sys.stderr.write(
➥ "reporter:counter:SkippingTaskCounters,MapProcessedRecords,1\n")
```

and the reduce task can use

```
sys.stderr.write(
➥ "reporter:counter:SkippingTaskCounters,ReduceProcessedGroups,1\n")
```

Java programs that cannot depend on the default record counting should use

```
context.getCounter(SkipBadRecords.COUNTER_GROUP,
      SkipBadRecords.COUNTER_MAP_PROCESSED_RECORDS).increment(1);
```

and

```
context.getCounter(SkipBadRecords.COUNTER_GROUP,
      SkipBadRecords.COUNTER_REDUCE_PROCESSED_GROUPS).increment(1);
```

when it has processed a key/value pair in its `Mapper` and `Reducer`, respectively.

7.2.3 Rerunning failed tasks with IsolationRunner

Debugging through log files is about reconstructing events using generic historical records. Sometimes there's not enough information in the logs to trace back the cause of failure. Hadoop has an IsolationRunner utility that functions like a time machine for debugging. This utility can isolate and rerun the failed task with the exact same input on the same node. You can attach a debugger to monitor the task as it runs and focus on gathering evidence specific to the failure.

To use the IsolationRunner feature, you must run your job with the configuration property `mapreduce.task.files.preserve.failedtasks` set to true. This tells every MapReduce application to keep all the data necessary to rerun the failed tasks.

When a job fails, you use the ResourceManager web UI to locate the node, the job ID, and the task attempt ID of the failed task. You log into the node where the task failed and go to the work directory under the directory for the task attempt. Go to

```
local_dir/taskTracker/jobcache/job_id/attempt_id/work
```

where *job_id* and *attempt_id* are the job ID and task attempt ID of the failed task. (The job ID should start with "job_" and the task attempt ID should start with "attempt_".) The root directory *local_dir* is what is set in the configuration property `mapreduce.cluster.local.dir`. Note that Hadoop allows a node to use multiple local directories (by setting `mapreduce.cluster.local.dir` to a comma-separated list of directories) to spread out disk I/O among multiple drives. If the node is configured that way, you'll have to look in all the local directories to find the one with the right attempt_id subdirectory.

Within the work directory you can execute IsolationRunner to rerun the failed task with the same input that it had before. In the rerun, we want the JVM to be enabled for remote debugging. As we're not running the JVM directly but through the bin/hadoop script, we specify the JVM debugging options through `HADOOP_OPTS`:

```
export HADOOP_OPTS="-agentlib:jdwp=transport=dt_socket,
➥ server=y,address=8000"
```

It tells the JVM to listen for the debugger at port 8000 and to wait for the debugger getting attached before running any code.[20] We now use IsolationRunner to rerun the task:

```
bin/hadoop org.apache.hadoop.mapred.IsolationRunner ../job.xml
```

The job.xml file contains all the configuration information IsolationRunner needs. Given our specification, the JVM will wait for a debugger's attachment before executing the task. You can attach to the JVM any Java debugger that supports the Java Debug Wire Protocol (JDWP). All the major Java IDEs do so. For example, if you're using `jdb`, you can attach it to the JVM via

```
jdb -attach 8000
```

(Of course, this is only an example. I hope you're using something better than `jdb`!) Consult your IDE's documentation for how to connect its debugger to a JVM.

7.3 Tuning for performance

After you have developed your MapReduce program and fully debugged it, you may want to start tuning it for performance. Before doing any optimization, note that one of the main attractions of Hadoop is its linear scalability. You can speed up many jobs by adding more machines. This makes economic sense when you have a small cluster. Consider the value of time it takes to optimize your program to gain a 10 percent improvement. For a 10-node cluster, you can get the same 10 percent performance gain by adding one machine (and this gain applies to all jobs on that cluster). The cost of your development time may well be higher than the cost of the additional computer. On the other hand, for a 1,000-node cluster, squeezing a 10 percent improvement through hardware will take 100 new machines. At that

[20] Options to configure the Sun JVM for debugging are further explained in Sun's documentation: http://java.sun.com/javase/6/docs/technotes/guides/jpda/conninv.html#Invocation.

scale the brute force approach of adding hardware to boost performance may be less cost effective.

Hadoop has a number of specific levers and knobs for tuning performance, some of which boost the effectiveness of the cluster as a whole. We cover those in the next chapter when we discuss system administration issues. In this section we examine techniques that can be applied on a per-job basis.

7.3.1 Reducing network traffic with combiner

Combiner can reduce the amount of data shuffled between the map and reduce phases, and lower network traffic improves execution time. The details and the benefits of using combiner are thoroughly described in section 4.6. We mention it here again for the sake of completeness.

7.3.2 Reducing the amount of input data

When processing large data sets, sometimes a nontrivial portion of the processing time is spent scanning data from disk. Reducing the number of bytes to read can enhance overall throughput. There are several ways to do this.

The simplest way to reduce the amount of bytes processed is to reduce the amount of data processed. We can choose to process only a sampled subset of the data. This is a viable option for certain analytics applications. For those applications, sampling reduces precision but not accuracy. Their results remain useful for many decision support systems.

Often your MapReduce jobs don't use all the information in the input data set. Recall our patent description data set from chapter 4. It has almost a couple dozen fields, yet most of our jobs access only a few common ones. It's inefficient for every job on that data set to read the unused fields every time. One can "refactor" the input data into several smaller data sets. Each has only the fields necessary for a particular type of data processing. The exact refactoring will be application dependent. This technique is similar in spirit to vertical partitioning and column-oriented databases in the relational database management system (RDBMS) world.

Finally, you can reduce the amount of disk and network I/O by compressing your data. You can apply this technique to the intermediate as well as output data sets. Hadoop has many options for data compression, and we devote the next subsection to this topic.

7.3.3 Using compression

Even with the use of a combiner, the output of the map phase can be large. This intermediate data has to be stored on disk and shuffled across the network. Compressing this intermediate data will improve performance for most MapReduce jobs, and it's easy too.

Hadoop has built-in support for compression and decompression. Enabling compression on the mapper's output involves setting two configuration properties, as you can see in Table 7.3.

Table 7.3 Configuration properties to control the compression of mapper’s output

Property	description
mapreduce.map.output.compress	Boolean property denoting whether the output of mapper should be compressed
mapreduce.output.fileoutputformat.compress.codec	Class property denoting which `CompressionCodec` to use for compressing mapper’s output

To enable compression on the mapper’s output, you set `mapreduce.map.output.compress` to true. In addition, you should set `mapreduce.output.fileoutputformat.compress.codec` to the appropriate codec class. All codec classes in Hadoop implement the `CompressionCodec` interface. Hadoop supports a number of compression codecs (see Table 7.4). For example, to use GZIP compression, you can set the configuration object:

```
conf.setBoolean("mapreduce.map.output.compress", true);
conf.setClass("mapreduce.output.fileoutputformat.compress.codec",
              GzipCodec.class,
              CompressionCodec.class);
```

Table 7.4 List of codecs available under the `org.apache.hadoop.io.compress` package

Codec	Hadoop version	Description
`DefaultCodec`	0.18, 0.19, 0.20,2.4.0	Works with files in the zlib format. By Hadoop convention filenames for these files end in .deflate.
`GzipCodec`	0.18, 0.19, 0.20,2.4.0	Works with files in the gzip format. These files have a filename extension of .gz.
`BZip2Codec`	0.19, 0.20,2.4.0	Works with files in the bzip2 format. These files have a filename extension of .bz2. This compression format is unique in that it’s splittable for Hadoop, even when used outside the sequence file format.

Data output from the map phase of a job is used only internally to the job, so enabling compression for this intermediate data is transparent to the developer and is a no-brainer. As many MapReduce applications involve multiple jobs, it makes sense for jobs to be able to output and input in compressed form. It’s highly recommended that data that are passed between Hadoop jobs use the Hadoop-specific sequence file format.

Sequence file is a compressable binary file format for storing key/value pairs. It is designed to support compression while remaining splittable. Recall that one of the parallelisms of Hadoop is its ability to split an input file for reading and processing by multiple map tasks. If the input file is in a compressed format, Hadoop will have to be able to split the file such that each split can be decompressed by the map tasks independently. Otherwise parallelism is destroyed if Hadoop has to decompress the file as a whole first. Not all compressed file formats are designed for splitting and decompressing in chunks. Sequence files were specially developed to support this feature. The file format provides sync markers to Hadoop to denote splittable boundaries.[21]

In addition to its compressability and splittability, sequence files support binary keys and values. Therefore, a sequence file is often used for processing binary documents, such as images, and it works great for text documents and other large key/value objects as well. Each document is considered a record within the sequence file.

You can make a MapReduce job output a sequence file by setting its output format to `SequenceFileAsBinaryOutputFormat`. You'll want to change its compression type from the default `RECORD` to `BLOCK`. With record compression, each record is compressed separately. With block compression, a block of records is compressed together and achieves a higher compression ratio. Finally, you have to call the static methods `setCompressOutput()` and `setOutputCompressorClass()` in `FileOutputFormat` (or `SequenceFileOutputFormat`, which inherits those methods) to enable output compression using a specific codec. The supported codecs are the same as those given in Table 7.4. You add these lines to the driver:

```
job.setOutputFormatClass(SequenceFileAsBinaryOutputFormat.class);
      SequenceFileAsBinaryOutputFormat.setOutputCompressionType(job,
      CompressionType.BLOCK);
FileOutputFormat.setCompressOutput(job, true);
      FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);
```

Table 7.5 lists the equivalent properties for configuring for sequence file output. A Streaming program can output sequence files when given the following options:

```
-outputformat org.apache.hadoop.mapreduce.SequenceFileOutputFormat
-D mapred.output.compression.type=BLOCK
-D mapred.output.compress=true
-D mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCode
```

Table 7.5 Configuration properties for outputting compressed sequence file

Property	Description
`mapreduce.output.fileoutputformat.compress.type`	String property to denote the sequence file's compression type. Can be one of `NONE`, `RECORD`, or `BLOCK`. Default is `RECORD` but `BLOCK` almost always compresses better. **Convenience method:** `FileOutputFormat.setCompressOutput()`

[21] All the input files we've seen so far are uncompressed text files where each record is a line. The newline character `(\n)` can trivially be thought of as the sync marker pointing out to both splittable boundaries and record boundaries.

`mapreduce.output.fileoutputformat.compress`	Boolean property on whether to compress the job's output. **Convenience method:** `SequenceFileOutputFormat.` ➥ `setOutputCompressionType()`
`mapreduce.output.fileoutputformat.compress.codec`	Class property that is used to specify which compression codec to use for compressing the job's output. **Convenience method:** `FileOutputFormat.` ➥ `setOutputCompressorClass()`

To read a sequence file as input, set the input format to `SequenceFileInputFormat`. Use

```
job.setInputFormatClass(SequenceFileAsBinaryInputFormat.class);
```

or

```
-inputformat org.apache.hadoop.mapreduce.SequenceFileAsBinaryInputFormat
```

for Streaming. There's no need to configure the compression type or codec class, as the `SequenceFile.Reader` class (used by `SequenceFileRecordReader`) will automatically determine those settings from the file header.

7.3.4 Reusing the JVM

By default, the YARN ApplicationMaster runs each `Mapper` and `Reducer` task in a separate JVM as a child process. This necessarily incurs the JVM start-up cost for each task. If the mapper does its own initialization, such as reading into memory a large data structure (see the example of joining using distributed cache in section 5.2.2), that initialization is part of the start-up cost as well. If each task runs only briefly, or if the mapper initialization takes a long time, then the start-up cost can be a significant portion of a task's total run time.

Hadoop allows the reuse of a JVM across multiple tasks of the same job. The start-up cost can, therefore, be amortized across many tasks. The property, `mapreduce.job.jvm.numtasks`, specifies the maximum number of tasks (of the same job) a JVM can run. The default value is 1; JVM is not reused. You can enable JVM reuse by setting the property to a higher number. You can also set it to -1, which means there's no limit to the number of tasks a JVM can be reused for. This is summarized in Table 7.6.

Table 7.6 Configuration property for enabling JVM reuse

Property	**description**
`mapreduce.job.jvm.numtasks`	Integer property for setting the maximum number of tasks a JVM can run. A value of -1 means no limit.

7.3.5 Running with speculative execution

One of the original design assumptions of MapReduce (as stated in the Google MapReduce paper) is that nodes are unreliable and the framework must handle the situation where some nodes fail in the middle of a job. Under this assumption, the original MapReduce framework specifies the map tasks and the reduce tasks to be idempotent. This means that when a task fails, Hadoop can restart that task and the overall job will end up with the same result. Hadoop can monitor the health of running nodes and restart tasks on failed nodes automatically. This makes fault tolerance transparent to the developer.

Often nodes don't suddenly fail but experience slowdown as I/O devices go bad. In such situations everything works but the tasks run slower. Sometimes tasks also run slow because of temporary congestion. This doesn't affect the correctness of the running job but certainly affects its performance. Even one slow-running task will delay the completion of a MapReduce job. Until all mappers have finished, none of the reducers will start running. Similarly, a job is not considered finished until all the reducers have finished.

Hadoop uses the idempotency property again to mitigate the slow-task problem. Instead of restarting a task only after it has failed, Hadoop will notice a slow-running task and schedule the same task to be run in another node in parallel. Idempotency guarantees the parallel task will generate the same output. Hadoop will monitor the parallel tasks. As soon as one finishes successfully, Hadoop will use its output and kill the other parallel tasks. This entire process is called speculative execution.

Note that speculative execution of map tasks will take place only after all map tasks have been scheduled to run, and only for map tasks that are making much less progress than is average on the other map tasks. It's the same case for speculative execution of reduce tasks. Speculative execution does not "race" multiple copies of a task to get the best completion time. It only prevents the slow tasks from dragging down the job's completion time.

By default, speculative execution is enabled. One can turn it off for map tasks and reduce tasks separately. To do this, set one or both of the properties in Table 7.7 to false. They're applied on a per-job basis, but you can also change the cluster-wide default by setting them in the cluster configuration file.

Table 7.7 Configuration properties for enabling and disabling speculative execution

Property	Description
`mapreduce.map.speculative`	Boolean property denoting whether speculative execution is enabled for map tasks
`mapreduce.reduce.speculative`	Boolean property denoting whether speculative execution is enabled for reduce tasks

You should leave speculative execution on in general. The primary reason to turn it off is if your map tasks or reduce tasks have side effects and are therefore not idempotent. For example, if a task writes to external files, speculative execution can cause multiple copies of a task to collide in attempting to create the same external files. You can turn off speculative execution to ensure that only one copy of a task is being run at a time.

> **NOTE** If your tasks have side effects, you should also think through how Hadoop's recovery mechanism would interact with those side effects. For example, if a task writes to an external file, it's possible that the task dies right after writing to the external file. In that case, Hadoop will restart the task, which will try to write to that external file again. You need to make sure your tasks' operation remains correct in such situations.

7.3.6 Refactoring code and rewriting algorithms

If you're willing to rewrite your MapReduce programs to optimize performance, some straightforward techniques and some nontrivial, application-dependent rewritings can speed things up.

One straightforward technique for a Streaming program is to rewrite it for Hadoop Java. Streaming is great for quickly creating a MapReduce job for ad hoc data analysis, but it doesn't run as fast as Java under Hadoop. Streaming jobs that start out as one-off queries but end up being run frequently can gain from a Java re-implementation.

If you have several jobs that run on the same input data, there are probably opportunities to rewrite them into fewer jobs. For example, if you're computing the maximum as well as the minimum of a data set, you can write a single MapReduce job that computes both rather than compute them separately using two different jobs. This may sound obvious, but in practice many jobs are originally written to do one function well. This is a good design practice. A job's conciseness makes it widely applicable to different data sets for different purposes. Only after some usage should you start looking for job groupings that you can rewrite to be faster.

One of the most important things you can do to speed up a MapReduce program is to think hard about the underlying algorithm and see if a more efficient algorithm can compute the same results faster. This is true for any programming, but it is more significant for MapReduce programs. Standard textbooks on algorithm and data structure (sorting, lists, maps, etc.) comprehensively cover design choices for most traditional programming. Hadoop programs, on the other hand, tend to touch on "exotic" areas, such as distributed computing, functional programming, statistics, and data-intensive processing, where best practices are less known to most programmers and there is still exciting research today to explore new approaches.

One example we've already seen that leverages a new data structure to speed up MapReduce programs is the use of Bloom filters in semijoins (section 5.3). The Bloom filter is well known in the distributed computing community but relatively unknown outside of it.

Another classic example of using a new algorithm to speed up a MapReduce program comes from statistics in the calculation of variance. Non-statisticians may compute variance using its canonical definition:

$(1/N) * Sum_i[(X_i - X_{avg})^2]$

where Sum_i denotes summation over the data set. The variable X_{avg} is the average of the data set. If we don't know that average ahead of time, then a non-statistician may decide to run one MapReduce job to find the average first, and a second MapReduce job to compute the variance. Someone more familiar with computing statistics will use an equivalent definition:

$(1/N) * Sum_i[(X_i)^2] - ((1/N) * Sum_i[X_i])^2$

From this definition one needs the sum of X as well as the sum of X_2, but you can compute both sums together in one scan of the data, using only a single MapReduce job. (This is analogous to the example of calculating maximum and minimum in a single job.) A little statistical background has halved the processing time in computing variance.[22]

You should also pay attention to the computational complexity of your algorithms. Hadoop provides "only" linear scalability, and you can still bring it to its knees with large data sets running under computationally intensive algorithms that are quadratic or worse. You certainly should look for more efficient algorithms in those cases, and sometimes you may have to settle for faster algorithms that only give approximate results.

7.4 Summary

Development methodologies for Hadoop build on top of best practices for Java programming, such as unit testing and test-driven development. Hadoop's central role of processing data calls for more data-centric testing processes. Math and logic errors are more prevalent in data-intensive programs and they're often inconspicuous. The distributed nature of Hadoop makes debugging much harder. To lessen the burden, you should test in stages, from a nondistributed (i.e., local) mode to a single-node pseudo-distributed mode, and finally to a fully distributed mode.

The famous computer scientist Donald Knuth once said that "premature optimization is the root of all evil." You should tune your Hadoop program for performance only after it's been fully debugged. Beyond thinking through general algorithmic and computational issues, performance enhancement is platform-specific, and Hadoop has a number of specific techniques to make jobs run more efficiently.

[22] There's a lot of nuisance in numerical computation over large data. In this variance calculation example we note our refactored MapReduce job has lower numerical precision and is more likely to run into overflow problems.

8 Data Security for Data Management

In this chapter, you will learn about Hadoop data security technologies and methods. First you will learn how file-system security is handled in Hadoop, including groups and permissions. In this section we also cover the newer Access Control List (ACL) system that makes security more versatile. Next we cover LDAP and Kerberos security that have been effective in enterprise security environments for more than two decades. Microsoft security is now based on Kerberos, and Hadoop has incorporated Kerberos to provide coordinated protection among all of the parts of Hadoop. We then show how to set-up and use newer systems for managing security in Hadoop that cover single sign on (SSO) and fine-grained data access.

Security on Hadoop can be complicated because of the complex interactions between the different daemons and services involved in a Hadoop cluster. But the situation is improving through the efforts of the open source community and commercial contributors.

Let's start with the essentials of file system security and users.

8.1 HDFS Security

Hadoop security begins with securing HDFS. The security model for HDFS is based on the POSIX security model. In this model, access permissions are assigned to groups and users. Each file or directory has read and write permissions for its owner, group, or all users. The user identity is determined by the host operating system when Hadoop is in "simple" mode. If Kerberos support is turned on, the identity comes from the Kerberos credentials that may also be shared with LDAP. Note that there are no execute permissions in the HDFS security model because files cannot be executed; MapReduce programs execute from outside the HDFS file system.

The way that access is resolved is very simple. When a user tries to access a file, the NameNode uses native (typically Linux) commands to get the user's groups either through a JNI interface or by actually shelling out and executing the bash command *groups*. Thus the NameNode machine needs to know the complete user and group set when using the simple model. The user and groups for a file on HDFS are stored in the NameNode as strings.

When you start the NameNode, the user who starts it becomes the superuser for all of HDFS and will have complete access to all of the files. That user doesn't need to be root on the NameNode (and really shouldn't be for security purposes).

During Hadoop installation, a common strategy is to create several different users and groups for Hadoop. The hdfs user becomes the superuser for HDFS while the yarn user and mapred users own resource management and MapReduce process logging, respectively. Each user also has a group associated with it and these users own the corresponding file system resources on local disks across the cluster. Finally, a hadoop group provides for general access on HDFS but without a corresponding user. All Hadoop users should be in the hadoop group. Table shows a standard configuration:

Table 8.1 File System and HDFS directory ownership for standard configuration

User	Programs	File System Directories	Logging	HDFS Directories
hdfs	NameNode, DataNode, Secondary NameNode	dfs.namenode.name.dir dfs.datanode.data.dir	HDFS_LOG_DIR	/ (ALL)
yarn	ResourceManager, NodeManager	yarn.nodemanager.local-dirs yarn.nodemanager.log-dirs container-executor conf/container-executor.cfg	YARN_LOG_DIR	yarn.nodemanager.remote-app-log-dir
mapred	Job History Server		MAPRED_LOG_DIR	mapreduce.jobhistory.intermediate-done-dir mapreduce.jobhistory.done-dir

In addition to these basic services, each additional service like Hive, Tez, Pig, etc. should have its own user and will also leverage the hadoop group. All ordinary user of the Hadoop cluster will also need to be added to the hadoop group.

Finally, in Hadoop there is the idea of a supergroup that does not necessarily have a group affiliation on the file system. When the HDFS superuser (whoever starts the NameNode) initially formats HDFS, the group assignment for the root of HDFS will be assigned to *supergroup*. A new user who is a member of the hadoop group will not be able to create directories or files at the /-level of the file system. They probably shouldn't, either. The general best practice is to create user directories that they can access. However, you can also create different sets of privileged users, assigning them to a new group and then set the supergroup to that group. For example, let's say that you have two users who need to have HDFS superuser permissions. You can create a new Unix/Linux group, hdfssuper, and add the users to it:

```
# groupadd hdfssuper
# usermod -a -G hdfssuper myuser1
# usermod -a -G hdfssuper myuser2
```

Next, you will change the property dfs.permissions.supergroup int etc/hadoop/hdfs-site.xml to the new hdfssuper group that you created and restart the NameNode. Then both myuser1 and myuser2 will be able to act as the superuser. The files they create will still be labeled as being supergroup rather than their file system groups, but the *supergroup* designation will now be mapped to hdfssuper.

From a MapReduce programming perspective, the HDFS security model carries over just like in Unix and Linux. If you run a MapReduce program as a certain user, that program will only be able to read and write files, create directories, and so forth where it has permissions to do so. Since you may execute the program from a computer that is not even part of the Hadoop cluster, your user and groups system must be shared across all of the machines that are involved in the Hadoop environment. Direct HDFS programming operations using the DFS API in Hadoop will carry the user credentials as well.

8.2 ACLs and Security

Access Control Lists (ACLs) are now supported in Hadoop as of Apache Hadoop 2.4. ACLs are an extension to the basic user and group security model inherited from POSIX. ACLs fix a fundamental problem with the POSIX model: access patterns may not map cleanly to organizational structures.

Let's say that user jane owns a file that contains a list of customers for a sales territory. Jane wants to let others in that territory read the file but reserves the modify permissions to herself. That is easily done in the file systems permissions model by setting read and write access to jane while creating a group westcoast that includes the others sales personnel in her territory. The group permissions are then set to read-only.

Now things change and Jane needs to delegate change permissions to three new regional managers while still keeping read-only for the rest of the sales organization. There is really no way to do this because you can't have two groups have separate permissions for the file, and you don't want everyone to be able to read the file. Traditional fixes to this problem were to create a synthetic user who Jane and her managers could su to when needed, but this seems crude and unnecessarily complex.

ACLs solve this dilemma by supporting a list of permissions that are considered in order. On Hadoop, you can turn on ACL support with the following property in hdfs-site.xml:

```
<property>
     <name>dfs.namenode.acls.enabled</name>
     <value>true</value>
```

```
</property>
```

With this property enabled you can now set and retrieve the ACLs on a file. You can set the user, group and all access, but you can also deny access to a single user or a group, or can enable access to others.

The commands used to retrieve and view the ACL for a file or directory is getacl:

```
$ hdfs dfs -getfacl /sales-data.txt 14/12/24 13:40:46
# file: /sales-data.txt
# owner: mark
# group: supergroup
user::rw-
group::r—
other::---
```

In this example, the basic file system permissions are the only access patterns enabled. These are inherited from the basic security system. But we can easily add to the list. For instance, if we want to allow a group called sales1 to modify the file, we just need to add a permission for that group to have write access:

```
$ hdfs dfs -setfacl -m group:sales1:rw- /sales-data.txt
```

Note the –m for the ACL specification and the requirement that each bit (read, write, and execute) be specified or given as a – to indicate not set or remove permission. Now if we get the ACLs again:

```
hdfs dfs -getfacl /sales-data.txt
# file: /sales-data.txt
# owner: mark
# group: supergroup
user::rw-
group::r—
group:sales1:rw-
mask::rw-
other::---
```

The sales1 group has now been given read and write permission for the file. In addition, the keyword “default” can be prefixed to the ACL specification in the setfacl call. Default then applies to subdirectories and files in those subdirectories when working with changing the ACLs of files. Also, the setfacl and getfacl calls support the –R flag to provide recursive operations just like chmod and chgrp.

Taken together, the basic permissions system combined with the ACL overlay provides a thorough security and access management system that is comparable to Unix/Linux and Windows systems for security. What we’ve discussed so far does not integrate with other systems for user authentication and on-the-wire encryption. To get those capabilities, we need LDAP and Kerberos, the subjects of the next sections.

8.3 LDAP for Hadoop

Security and permissions in Linux are distributed in that the groups and users are resolved on each computer in the network subdomain and Hadoop cluster. This means that in order to create a user who can access all the machines in a large cluster, scripts or tools are needed that visit each computer and execute a shell command to add the user.

LDAP provides a centralized directory service for resolving user identities and for other purposes like organizational structures. For Hadoop, LDAP support can replace the mappings of user identities to permissions, even using LDAP capabilities that are shared with Active Directory on Windows computers.

Configuring LDAP for Hadoop can be challenging. If your organization has existing LDAP servers, the users and groups need to be entered into the directory. This is done using file that specifies the user identity, their organization unit, and other useful facts. The file format is the LDAP Data Interchange Format (LDIF) and entries look like this:

```
dn: cn=hdfs,ou=services,dc=my-company,dc=com
objectclass:top
objectclass: applicationProcess
objectclass:simpleSecurityObject
cn: hdfs
userPassword:hdfs-password
```

This file is then imported into the LDAP database using a tool like ldapadd on Linux using OpenLDAP:

```
ldapadd -f hadoop.ldif -D cn=manager,dc=hadoop,dc=my-company,dc=com -w hadoop
```

After setting up LDAP to provide the authorization, the next step is to set up Hadoop to use LDAP to do the authorization. You modify core-site.xml to change Hadoop’s method, adding a range of parameters to the file and restarting:

```
<property>
<name>hadoop.security.group.mapping</name>
<value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>
<property>
<name>hadoop.security.group.mapping.ldap.bind.user</name>
<value>cn=Manager,dc=my-company,dc=com</value>
</property>
<property>
<name>hadoop.security.group.mapping.ldap.bind.password</name>
<value>ldappassword</value>
</property>
<property>
<name>hadoop.security.group.mapping.ldap.url</name>
<value>ldap://localhost:389/dc=my-company,dc=com</value>
</property>
<property>
     <name>hadoop.security.group.mapping.ldap.base</name>
     <value></value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.user</name>

      <value>(&amp;(|(objectclass=person)(objectclass=applicationProcess))(cn={0}))</
      value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.search.filter.group</name>
  <value>(objectclass=groupOfNames)</value>
</property>
<property>
     <name>hadoop.security.group.mapping.ldap.search.attr.member</name>
     <value>member</value>
</property>
<property>
       <name>hadoop.security.group.mapping.ldap.search.attr.group.name</name>
       <value>cn</value>
</property>
```

The critical components of this configuration should be familiar to anyone who deals with LDAP regularly. The most important property is `hadoop.security.group.mapping.ldap.url` that provides the URL of the LDAP server. In a typical Hadoop deployment that is secured with LDAP, you have to configure the firewall to communicate with the LDAP server.

8.4 Kerberos and Hadoop

Kerberos is a security system that not only authenticates but also provides secure communications between clients and servers. Kerberos is a proven system for security that is widely used. Windows security is based on Kerberos, for example. The Kerberos protocol is designed to avoid some of the challenges with traditional security mechanisms, but it was also designed primarily for two-tiered architectures where a client authenticates to a server and then is allowed to communicate with the server for a session. Hadoop has many different programs working together, and so authenticating to the ResourceManager to run a job also needs to support carrying that authorization through to the NameNode for access to HDFS resources.

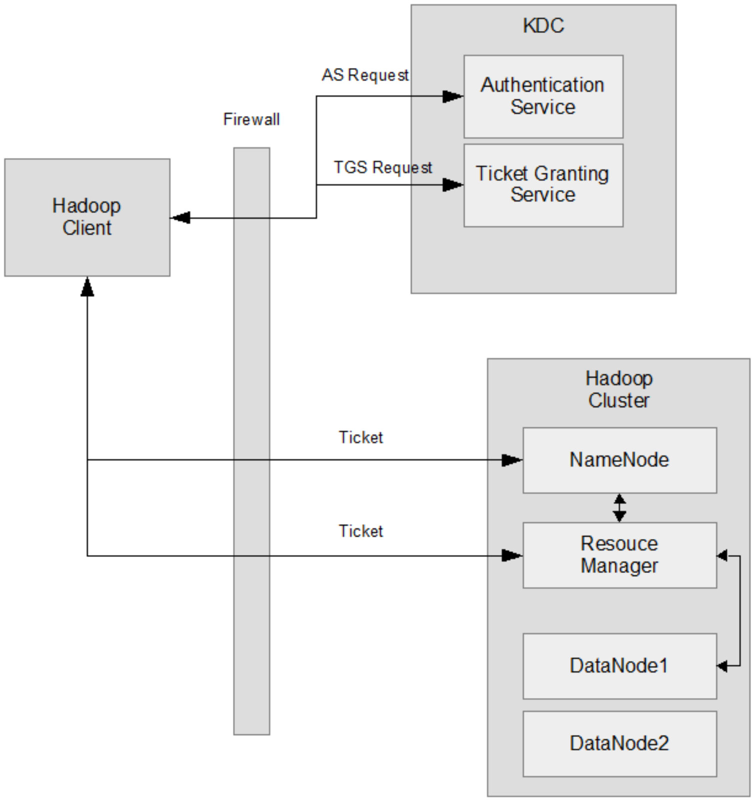


Figure 8.1 Hadoop client interacting with Kerberos-secured Hadoop. The KDC provides authentication and tickets for access to the Hadoop cluster.

With Kerberos, the client program does not transmit the user password to the Kerberos Authentication Server (AS) but instead creates a one-way hash out of it using the agreed to protocol. The client program does transmit the user identifier to the AS without bothering to encrypt it or the communications channel. The AS then encrypts a session key (if the user exists) using the user password as the key and returns that key along with an encrypted additional communication session key that will be used for the next phase of communications between the client and another Kerberos server called the Ticket Granting Server (TGS). If the client program can't decrypt the session key, the user entered the wrong password. The protocol then continues to issue "tickets" that authenticate to the individual services from the client. Both the AS and the TGS are part of the Key Distribution Center (KDC). **Error! Reference source not found.** shows the interactions between clients and the different parts of the KDC, as well as with ticketed access to Hadoop.

In Hadoop, additional tickets need to be proxied between the individual programs needed to execute actions over Hadoop, but this generally happens transparently in the background when running in secure mode. Configuring secure mode is a bit complex, but perhaps less daunting than managing the LDAP system without special tools. The major Hadoop vendors have also improved the installation and maintenance of Kerberos with Hadoop. Getting started with Kerberos and Apache requires installing Kerberos and configuring it, then configuring Hadoop to use Kerberos.

8.4.1 Getting and Installing Kerberos

All of the major Linux distributions have Kerberos available for installation. Before starting installation, you should insure that your cluster has fully-qualified domain names (FQDNs) for each node in the cluster. That means assigning each a resolvable host and domain as well as providing some kind of DNS services for them. For our example installation, we will use CentOS 6 running in a virtualization environment. By default, the system just installs as localhost and needs a FQDN for Kerberos to work with. To change this, you modify /etc/hosts as root and set HOSTNAME /etc/sysconfig/network. You can either reboot or restart networking to have the change take effect. For our example, we have assigned hadoop.example.com as the FQDN.

You will need to install several components of Kerberos as root:

```
# yum install krb5-libs krb5-server krb5-workstation
```

After installation, you create the initial configuration database for Kerberos:

```
# kdb5_util create -s
```

Then you will need to modify the configuration files to make your hostname the primary "realm" of Kerberos system. In Kerberos, credentials are associated with a principal, an instance, and a realm. For most Linux applications—including Hadoop—the principal will be the user name and the realm will be mapped to the FQDN, hadoop.example.com in our case.

The Kerberos configuration files are divided into two parts: /etc/krb5.conf and /var/kerberos/krb5kdc. In /etc/krb5.conf are the mappings between the realm and the hostname. In this file, realms are always in uppercase while domain names are in lower:

```
[logging]
default = FILE:/var/log/krb5libs.log
kdc = FILE:/var/log/krb5kdc.log
admin_server = FILE:/var/log/kadmind.log

[libdefaults]
default_realm = EXAMPLE.COM
dns_lookup_realm = false
dns_lookup_kdc = false
ticket_lifetime = 24h
renew_lifetime = 7d
forwardable = true

[realms]
EXAMPLE.COM = {
     kdc = hadoop.example.com
     admin_server = hadoop.example.com
}

[domain_realm]
hadoop.example.com = EXAMPLE.COM
```

Note the mappings both from realm to servers (The KDC is the Key Distribution Center and comprises the Authentication and Ticket Granting servers).

Next, /var/kerberos/krb5kdc/kadm.acl needs to be modified to map the admin principal to the correct realm:

```
/admin@EXAMPLE.COM     *
```

Once completed, start the Kerberos daemons:

```
# /sbin/service krb5kdc start
# /sbin/service kadmin start
```

At this point, you need to create the principals for your different user identities. You can do that with the kadmin.local command-line client, like this:

```
# kadmin.local
addprinc root/admin
addprinc hdfs/admin
```

Each time you will need to provide the admin password that you originally created when you created the Kerberos database. You will also need to provide new passwords for the users. It is possible to combine the Kerberos password authentication with LDAP but we will not cover that here.

You can test the ticket granting service from the command line using kinit. After getting a ticket, you can list your tickets using klist. The final step is in configuring Hadoop to use Kerberos. This involves modifying Hadoop etc/hadoop/core-site.xml for all the nodes in the cluster as well as generating "keytab" files for each host and each service on each host. A keytab file contains the principal and encrypted keys for a given service. This can be time consuming and it is helpful to create some automation to help with the task.

For Hadoop, there need to be keytab files on each cluster host for the different principals needed for the configuration. A typical set of principals is shown in Table.

Table 8.1 Kerberos principals for Hadoop configurations

Host	Principal
NameNode	hdfs
Secondary NameNode	hdfs
DataNode	Hdfs, mapred
ResourceManager	yarn
HTTP services	HTTP

You can create these principals and generate the keytab files using Kerberos utilities:

```
# kadmin.local
kadmin.local: addprinc -randkey hdfs/hadoop.example.com@EXAMPLE.COM

WARNING: no policy specified for hdfs/hadoop.example.com@EXAMPLE.COM; defaulting to
      no policy
Principal "hdfs/hadoop.example.com@EXAMPLE.COM" created.

kadmin.local:  xst -norandkey -k hdfs.keytab hdfs/hadoop.example.com

Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type aes256-cts-
      hmac-sha1-96 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type aes128-cts-
      hmac-sha1-96 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type des3-cbc-
      sha1 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type arcfour-hmac
      added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type des-hmac-
      sha1 added to keytab WRFILE:hdfs.keytab.
Entry for principal hdfs/hadoop.example.com with kvno 1, encryption type des-cbc-md5
      added to keytab WRFILE:hdfs.keytab.
```

Once all of the keytab files and principals have been created, they should be moved to the etc/hadoop directory and their permissions set to prevent reading by any other user. Finally, it is a simple matter to set Hadoop to secure mode and restart the cluster. Add the following to etc/hadoop/core-site.xml on every node:

```
<property>
  <name>hadoop.security.authentication</name>
  <value>kerberos</value>
</property>

<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
```

Finally, YARN and HDFS security need to be configured in etc/hadoop/hdfs-site.xml and etc/hadoop/yarn-site.xml. The following parameters are for HDFS and DataNode security:

```
<!-- HDFS Kerberos config -->
<property>
  <name>dfs.block.access.token.enable</name>
  <value>true</value>
</property>

<!-- NameNode Kerberos config -->
<property>
  <name>dfs.namenode.keytab.file</name>
  <value>/etc/hadoop/ hdfs.keytab</value> <!-- path to the HDFS keytab -->
</property>
<property>
  <name>dfs.namenode.kerberos.principal</name>
  <value>hdfs/hadoop.example.com@EXAMPLE.COM</value>
</property>
```

```
<property>
  <name>dfs.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/hadoop.example.com@EXAMPLE.COM</value>
</property>

<!-- Secondary NameNode Kerberos config -->
<property>
  <name>dfs.secondary.namenode.keytab.file</name>
  <value>/etc/hadoop/hdfs.keytab</value> <!-- path to the HDFS keytab -->
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.principal</name>
  <value>hdfs/hadoop.example.com@EXAMPLE.COM </value>
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/hadoop.example.com@EXAMPLE.COM</value>
</property>

<!-- DataNode Kerberos config -->
<property>
  <name>dfs.datanode.data.dir.perm</name>
  <value>700</value>
</property>
<property>
  <name>dfs.datanode.address</name>
  <value>0.0.0.0:1004</value>
</property>
<property>
  <name>dfs.datanode.http.address</name>
  <value>0.0.0.0:1006</value>
</property>
<property>
  <name>dfs.datanode.keytab.file</name>
  <value>/etc/hadoop/hdfs.keytab</value>
</property>
<property>
  <name>dfs.datanode.kerberos.principal</name>
  <value>hdfs/hadoop.example.com@EXAMPLE.COM</value>
</property>
```

And likewise for YARN:

```
<!-- ResourceManager Kerberos config -->
<property>
  <name>yarn.resourcemanager.keytab</name>
  <value>/etc/hadoop/yarn.keytab</value>
</property>
<property>
  <name>yarn.resourcemanager.principal</name>
  <value>yarn/hadoop.example.com@EXAMPLE.COM</value>
</property>

<!-- NodeManager Kerberos config -->
<property>
  <name>yarn.nodemanager.keytab</name>
  <value>/etc/hadoop/yarn.keytab</value>
</property>
<property>
  <name>yarn.nodemanager.principal</name>
  <value>yarn/hadoop.example.com@EXAMPLE.COM </value>
```

```
</property>
<property>
  <name>yarn.nodemanager.container-executor.class</name>
  <value>org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor</value>
</property>
<property>
  <name>yarn.nodemanager.linux-container-executor.group</name>
  <value>yarn</value>
</property>
```

Finally, additional configuration is needed for the mapred principal in etc/hadoop/mapred-site.xml:

```
<!-- MapReduce Job History Server Kerberos config -->
<property>
  <name>mapreduce.jobhistory.address</name>
  <value>hadoop.example.com:10020</value>
</property>
<property>
  <name>mapreduce.jobhistory.keytab</name>
  <value>/etc/hadoop/mapred.keytab</value>
</property>
<property>
  <name>mapreduce.jobhistory.principal</name>
  <value>mapred/hadoop.example.com@EXAMPLE.COM</value>
</property>
```

There is also one final configuration requirement for etc/hadoop/container-executor.cfg. Set the container executor group to yarn and add the users hdfs, mapred, yarn, etc. to the banned users. This configuration file governs what users can execute containers and only members of the yarn group. You should also set the min.user.id to 500 if you are working on CentOS.

And after all that work, you can restart your cluster and start testing functionality. Don't forget to check out your log files for both Hadoop and Kerberos (/var/log/krb5kdc.log) if things are working correctly.

8.5 Apache Knox

In the modern world of Internet services all of us have many passwords. Security professionals warn us not to use the same password at different sites, to change our passwords often, and to only use complex passwords that are hard to guess. All of that is good advice, but it makes remembering and managing passwords difficult. Single Sign On (SSO) in the enterprise helps to make access to different internal services and sites much easier. You log in once and your credentials are shared across all the services you are allowed to access.

Apache Knox provides SSO for many of the important services in Hadoop. You log in once using HTTPS and then can access different services using a REST API that delivers results in JSON, XML, or text. Applications can do the same thing. Knox also integrates with LDAP and Kerberos to provide authentication for both users and encrypted internal communications between Knox and the cluster.

Figure 8.2 shows Apache Knox as the gateway to using Hadoop services. The users and applications all communicate over HTTPS to ask for execution of services using REST calls. The results of the request are returned in JSON. Not all services are yet covered by Knox, but most of the core capabilities are there. You can manage HDFS through the WebHDFS proxy, execute Oozie workflows, interact with HCatalog and Hive, and work with HBase.

The Apache Knox package contains a sample LDAP server that is configured with an LDIF file, conf/users.ldif. Here's an example entry in the file:

```
dn: uid=guest,ou=people,dc=hadoop,dc=apache,dc=org
objectclass:top
objectclass:person
objectclass:organizationalPerson
objectclass:inetOrgPerson
cn: Guest
sn: User
uid: guest
userPassword:guest-password
```

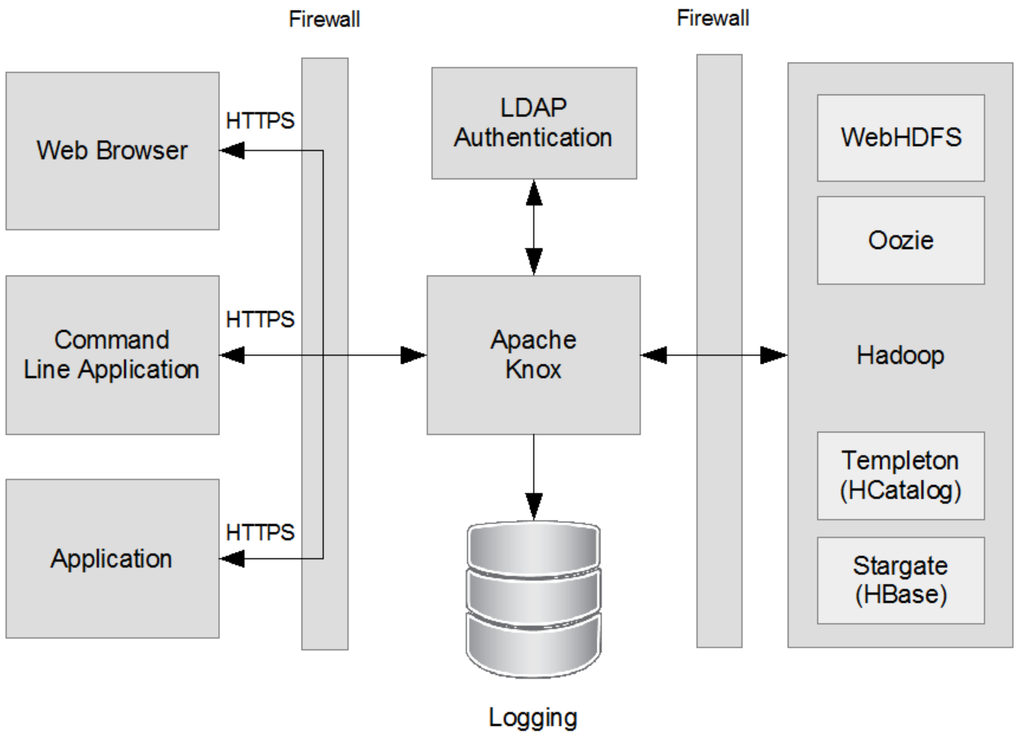


Figure 8.2 Apache Knox as part of an SSO-enabled secure system.

We will use this user identity and password in the examples to come. To start the LDAP server, use `bin/ldap.sh start`. By default, the Knox gateway will use the local LDAP server and default realms, but you can change those in the conf/topologies directory after initial configuration. Finally, you will need to create a master password for Knox to encrypt passwords. To do this, use the following command:

```
$ bin/knowcli.sh create-master
```

It's good practice for real deployments to create a unique user like "knox" and lock down the directories from read-access from other users. After creating a password, start Knox with:

```
$ bin/gateway.sh start
```

Knox will begin listening on port 8443 of the localhost. It will try to communicate with Hadoop by default on the localhost as well. Changing the ports and the service URLs is accomplished by modifying the configuration files in `conf/topologies`. Specifically, `sandbox.xml` contains the URLs for the NameNode, ResourceManager, WebHDFS node, etc.

After starting Knox, you can test that Knox works by communicating with WebHDFS to check out files and directories using REST calls. The best way to do that is using cURL:

```
$ curl -i -k -u guest:guest-password -X GET
      'https://localhost:8443/gateway/sandbox/webhdfs/v1/test?op=LISTSTATUS'
```

Here we are submitting a GET request to Knox using the guest user and that user's password defined in the users.ldif file, guest-password. The request is exercising one of Knox's services that are registered with it, webhdfs access. The URL for the request is HTTPS and then follows a REST-style convention. The gateway has several topologies registered, one of which is "sandbox", and sandbox has a webhdfs service with version "v1". The individual services then have distinct commands available to them. In our URL we are requesting a listing of the /test directory on HDFS. We can also retrieve a file from HDFS directly using a cURL request like this:

```
curl -i -k -u guest:guest-password -L
      'https://localhost:8443/gateway/sandbox/webhdfs/v1/test/mytext.txt?op=OPEN'
```

Here we are using the OPEN command to grab the file and the –L switch for cURL that forces cURL to redo the request on a redirect. Here we have retrieved mytext.txt to the console.

Knox also supports executing jobs using YARN and Oozie, the workflow manager for Hadoop. To use Oozie, you create an XML plan for the workflow and submit it using the gateway. Oozie requires a separate installation for the Apache version of Hadoop. You can get it from oozie.apache.org. It also needs to be built using Maven, itself available from maven.apache.org. Once Maven is installed, you can build Oozie using bin/mkdistro.sh in the Oozie directory. After building Oozie, you will need to modify your Hadoop configuration to allow the Oozie proxy to interact with it. Similar capabilities are available for HBase (via the Starbase interfaces) and HCatalog (via the Templeton project).

There are often several ways to achieve the same result. Both HCatalog and Oozie can be used to submit MapReduce jobs, for instance. The Oozie workflow capabilities make it the better choice for complex workflows, but the HCatalog approach can be used for quick-and-dirty submissions. Almost any programming language that supports HTTPS interactions can serve as a client for the Knox gateway, eliminating the strong dependencies on Java in the native Hadoop APIs and client libraries for the individual services. The JSON that is returned by default is also easily consumed by JavaScript services for display and visualization in a web tier.

8.6 Apache Sentry

So far we have seen how to secure the Hadoop filesystem, encrypt traffic, authenticate using LDAP and Kerberos, and use an SSO gateway to access key services. When treating Hadoop as a database, however, some of the finer grain security capabilities of traditional RDBMSs are lacking. There is no way to control access to parts of files that participate in Hive schema, for instance; only file-level security is available.

Enter Apache Sentry, which is still incubating as I write this. Sentry supports controlling access to tables in HCatalog, Hive, and Impala. It does this using policy files that define who can access what using Hive. A Hive user can therefore be given access to tables and parts of tables. Let’s say you are project lead for a telecommunications analysis project that needs to examine call patterns in an area. The data is arriving in Hadoop and is schematized using Hive. You need to be able to insert and modify data but don’t want your analysts to be able to do so. They only need to be able to read from it using select actions. Here’s an example of a policy file for Sentry:

```
[groups]
lead = cdr_insert_role, cdr_select_role
analyst = cdr_select_role

[roles]
cdr_insert_role = server=hadoop.example.com->db=cdr->table=*->action=insert
cdr_select_role = server=hadoop.example.com->db=cdr->table=*->action=select
```

Note that the file contains two blocks of configuration, groups and roles. The groups block maps Hadoop user groups to the roles. It’s assumed that the groups lead and analyst are Linux and Hadoop groups. What groups can access are defined in the roles block. For role definitions, Sentry supports server, uri, database, and table scoping. In our example, we are scoping the Hive server on our example VM, and then down to the level of inserts and selects over the CDR database. Any table may be read by the cdr_select_role, for instance.

Getting Sentry up and running requires installing Hive, Sentry and configuring the system for security. We will see how to do this in the next chapter on data access with Hive.

8.7 Overview

Security in Hadoop has come a long way from its early days when file-system security was the only way to secure your cluster. Managing access to the cluster was always a challenge, funneling users through the cluster managers who were simultaneously trying to learn how to keep a new, complex technology running and healthy. That's changed in recent years, with the integration of a rich set of enterprise security technologies as we've described here. Security remains an area of active development by the Hadoop community, with new challenges like in-memory computing creating new security challenges, but the core capabilities now are sophisticated enough that security-conscience organizations are able to use Hadoop without compromising their IR infrastructure or needing to wall-off Hadoop clusters as science experiments.