Part 1

Background and fundamentals

Part 1 of this book contains chapter 1, which looks at Hadoop's components and its ecosystem. The chapter then provides instructions for installing a pseudo-distributed Hadoop setup on a single host, and includes a system for you to run all of the examples in the book. Chapter 1 also covers the basics of Hadoop configuration, and walks you through how to write and run a MapReduce job on your new setup.

1 Hadoop in a heartbeat

This chapter covers

- Understanding the Hadoop ecosystem
- Downloading and installing Hadoop
- Running a MapReduce job

We live in the age of big data, where the data volumes we need to work with on a day-to-day basis have outgrown the storage and processing capabilities of a single host. Big data brings with it two fundamental challenges: how to store and work with voluminous data sizes, and more important, how to understand data and turn it into a competitive advantage.

Hadoop fills a gap in the market by effectively storing and providing computational capabilities over substantial amounts of data. It's a distributed system made up of a distributed filesystem and it offers a way to parallelize and execute programs on a cluster of machines (see figure 1.1). You've most likely come across Hadoop as it's been adopted by technology giants like Yahoo!, Facebook, and Twitter to address their big data needs, and it's making inroads across all industrial sectors.

Because you've come to this book to get some practical experience with Hadoop and Java, I'll start with a brief overview and then show you how to install Hadoop and run a MapReduce job. By the end of this chapter you'll have received

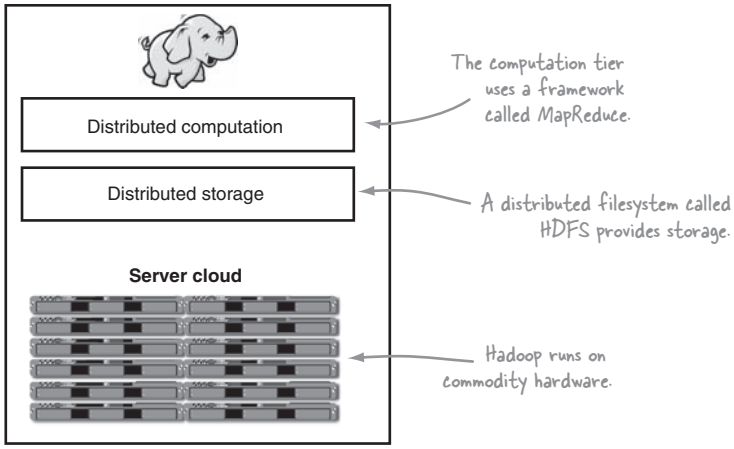


Figure 1.1 The Hadoop environment

a basic refresher on the nuts and bolts of Hadoop, which will allow you to move on to the more challenging aspects of working with Hadoop.[1]

Let's get started with a detailed overview of Hadoop.

1.1 What is Hadoop?

Hadoop is a platform that provides both distributed storage and computational capabilities. Hadoop was first conceived to fix a scalability issue that existed in Nutch,[2] an open source crawler and search engine. At the time Google had published papers that described its novel distributed filesystem, the Google File System (GFS), and MapReduce, a computational framework for parallel processing. The successful implementation of these papers' concepts in Nutch resulted in its split into two separate projects, the second of which became Hadoop, a first-class Apache project.

In this section we'll look at Hadoop from an architectural perspective, examine how industry uses it, and consider some of its weaknesses. Once we've covered Hadoop's background, we'll look at how to install Hadoop and run a MapReduce job.

Hadoop proper, as shown in figure 1.2, is a distributed master-slave architecture[3] that consists of the Hadoop Distributed File System (HDFS) for storage and MapReduce for computational capabilities. Traits intrinsic to Hadoop are data partitioning and parallel computation of large datasets. Its storage and computational capabilities scale with the addition of hosts to a Hadoop cluster, and can reach volume sizes in the petabytes on clusters with thousands of hosts.

In the first step in this section we'll examine the HDFS and MapReduce architectures.

---

[1] Readers should be familiar with the concepts provided in Manning's *Hadoop in Action* by Chuck Lam, and *Effective Java* by Joshua Bloch.

[2] The Nutch project, and by extension Hadoop, was led by Doug Cutting and Mike Cafarella.

[3] A model of communication where one process called the master has control over one or more other processes, called slaves.

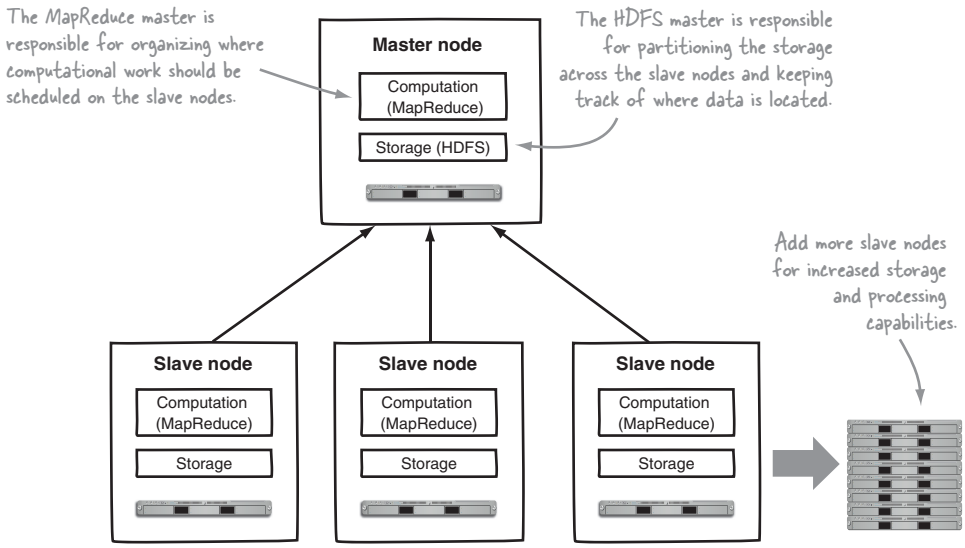


Figure 1.2 High-level Hadoop architecture

1.1.1 Core Hadoop components

To understand Hadoop's architecture we'll start by looking at the basics of HDFS.

HDFS

HDFS is the storage component of Hadoop. It's a distributed filesystem that's modeled after the Google File System (GFS) paper.[4] HDFS is optimized for high throughput and works best when reading and writing large files (gigabytes and larger). To support this throughput HDFS leverages unusually large (for a filesystem) block sizes and data locality optimizations to reduce network input/output (I/O).

Scalability and availability are also key traits of HDFS, achieved in part due to data replication and fault tolerance. HDFS replicates files for a configured number of times, is tolerant of both software and hardware failure, and automatically re-replicates data blocks on nodes that have failed.

Figure 1.3 shows a logical representation of the components in HDFS: the NameNode and the DataNode. It also shows an application that's using the Hadoop filesystem library to access HDFS.

Now that you have a bit of HDFS knowledge, it's time to look at MapReduce, Hadoop's computation engine.

MAPREDUCE

MapReduce is a batch-based, distributed computing framework modeled after Google's paper on MapReduce.[5] It allows you to parallelize work over a large amount of

[4] See the Google File System, http://research.google.com/archive/gfs.html.

[5] See MapReduce: Simplified Data Processing on Large Clusters, http://research.google.com/archive/mapreduce.html.

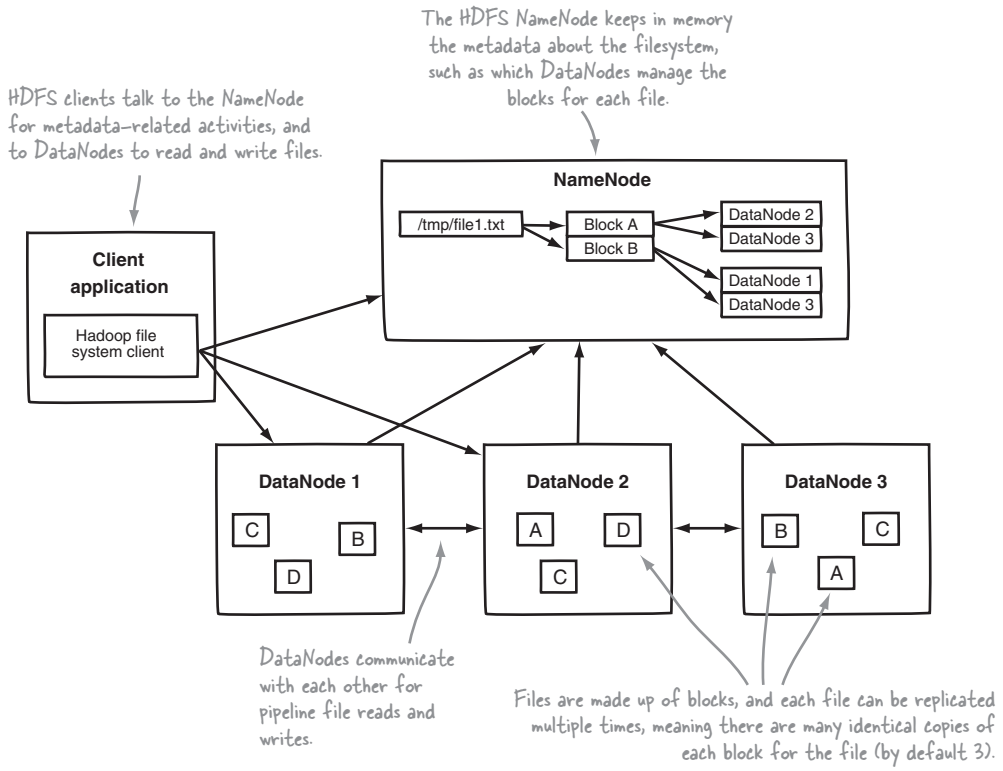


Figure 1.3 HDFS architecture shows an HDFS client communicating with the master NameNode and slave DataNodes.

raw data, such as combining web logs with relational data from an OLTP database to model how users interact with your website. This type of work, which could take days or longer using conventional serial programming techniques, can be reduced down to minutes using MapReduce on a Hadoop cluster.

The MapReduce model simplifies parallel processing by abstracting away the complexities involved in working with distributed systems, such as computational parallelization, work distribution, and dealing with unreliable hardware and software. With this abstraction, MapReduce allows the programmer to focus on addressing business needs, rather than getting tangled up in distributed system complications.

MapReduce decomposes work submitted by a client into small parallelized map and reduce workers, as shown in figure 1.4. The map and reduce constructs used in MapReduce are borrowed from those found in the Lisp functional programming language, and use a shared-nothing model[6] to remove any parallel execution interdependencies that could add unwanted synchronization points or state sharing.

[6] A shared-nothing architecture is a distributed computing concept that represents the notion that each node is independent and self-sufficient.

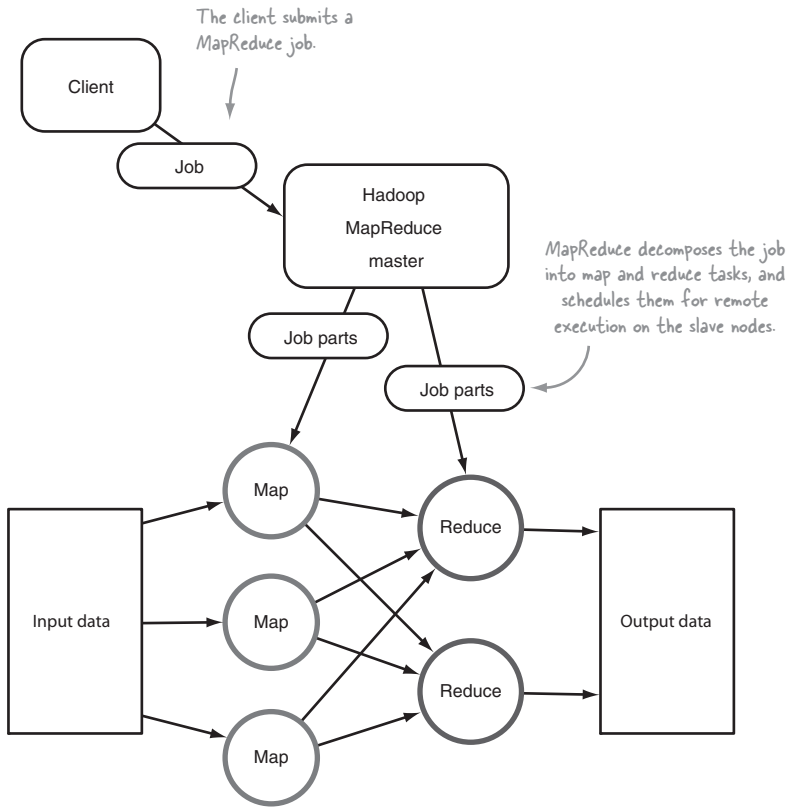


Figure 1.4 **A client submitting a job to MapReduce**

The role of the programmer is to define map and reduce functions, where the map function outputs key/value tuples, which are processed by reduce functions to produce the final output. Figure 1.5 shows a pseudo-code definition of a map function with regards to its input and output.

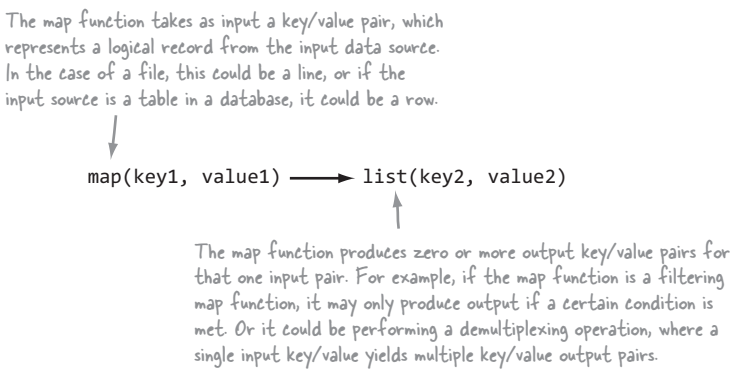


Figure 1.5 **A logical view of the map function**

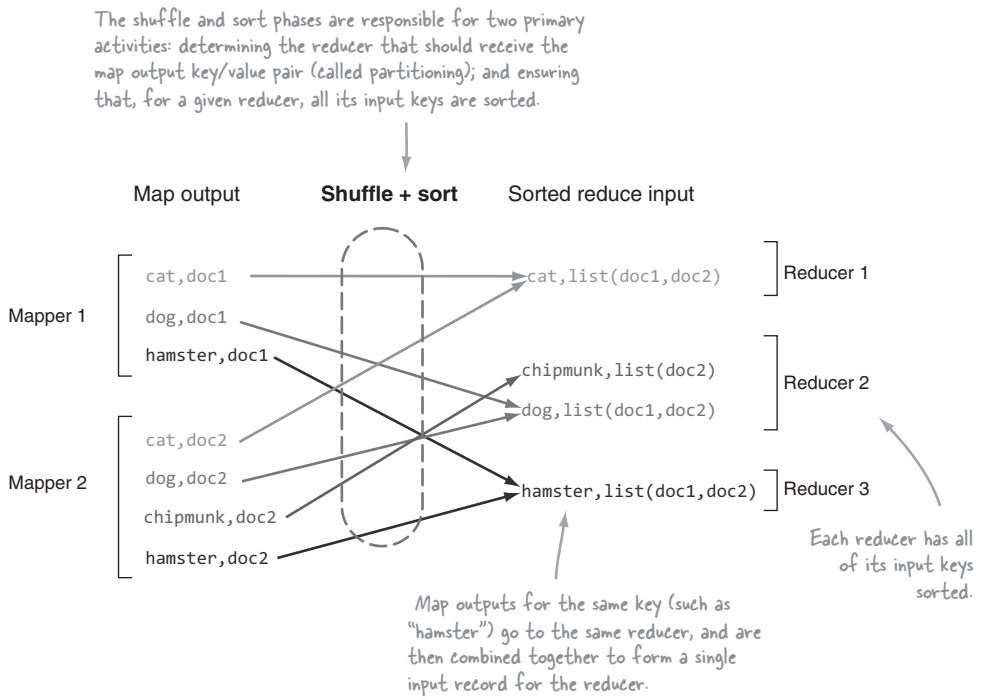


Figure 1.6 MapReduce's shuffle and sort

The power of MapReduce occurs in between the map output and the reduce input, in the shuffle and sort phases, as shown in figure 1.6.

Figure 1.7 shows a pseudo-code definition of a reduce function.

Hadoop's MapReduce architecture is similar to the master-slave model in HDFS. The main components of MapReduce are illustrated in its logical architecture, as shown in figure 1.8.

With some MapReduce and HDFS basics tucked under your belts, let's take a look at the Hadoop ecosystem, and specifically, the projects that are covered in this book.

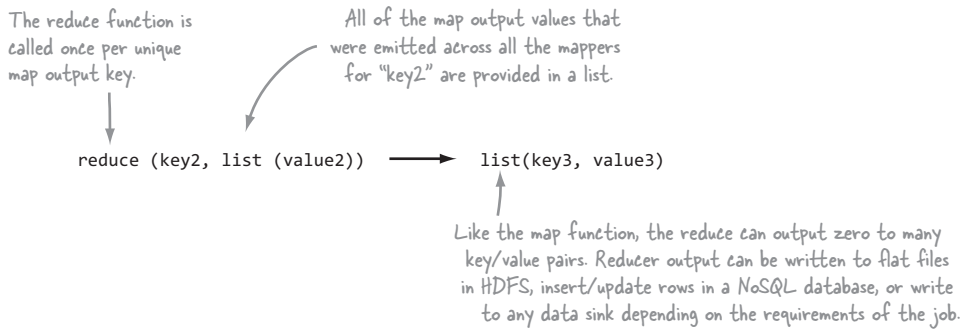


Figure 1.7 A logical view of the reduce function

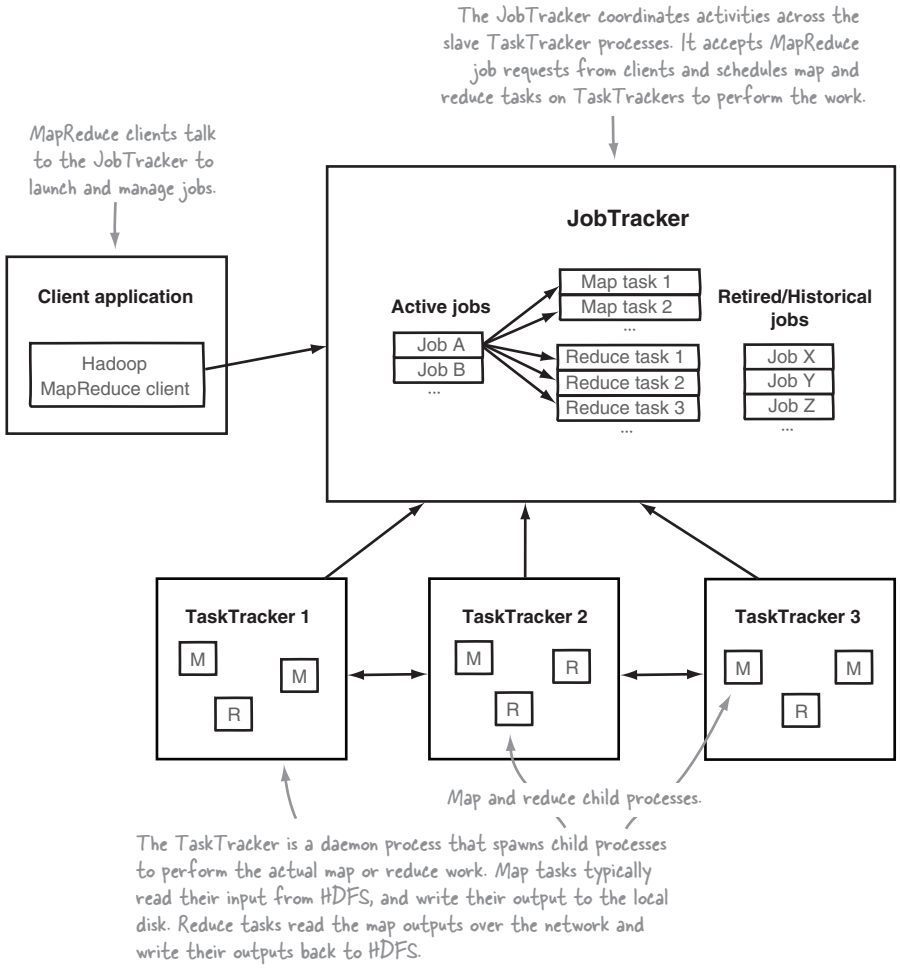


Figure 1.8 **MapReduce logical architecture**

1.1.2 The Hadoop ecosystem

The Hadoop ecosystem is diverse and grows by the day. It's impossible to keep track of all of the various projects that interact with Hadoop in some form. In this book the focus is on the tools that are currently receiving the greatest adoption by users, as shown in figure 1.9.

MapReduce is not for the faint of heart, which means the goal for many of these Hadoop-related projects is to increase the accessibility of Hadoop to programmers and nonprogrammers. I cover all of the technologies listed in figure 1.9 in this book and describe them in detail within their respective chapters. In addition, I include descriptions and installation instructions for all of these technologies in appendix A.

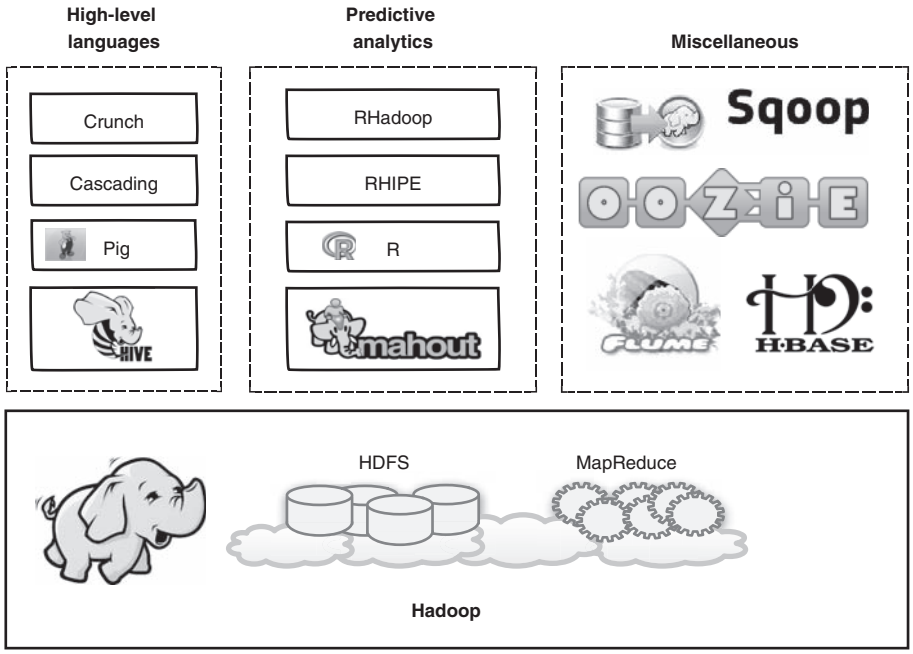


Figure 1.9 Hadoop and related technologies

Let's look at how to distribute these components across hosts in your environments.

1.1.3 Physical architecture

The physical architecture lays out where you install and execute various components. Figure 1.10 shows an example of a Hadoop physical architecture involving Hadoop and its ecosystem, and how they would be distributed across physical hosts. ZooKeeper requires an odd-numbered quorum,[7] so the recommended practice is to have at least three of them in any reasonably sized cluster.

For Hadoop let's extend the discussion of physical architecture to include CPU, RAM, disk, and network, because they all have an impact on the throughput and performance of your cluster.

The term *commodity hardware* is often used to describe Hadoop hardware requirements. It's true that Hadoop can run on any old servers you can dig up, but you still want your cluster to perform well, and you don't want to swamp your operations department with diagnosing and fixing hardware issues. Therefore, commodity refers to mid-level rack servers with dual sockets, as much error-correcting RAM as is affordable, and SATA drives optimized for RAID storage. Using RAID, however, is strongly discouraged on the DataNodes, because HDFS already has replication and error-checking built-in; but on the NameNode it's strongly recommended for additional reliability.

[7] A quorum is a High Availability (HA) concept that represents the minimum number of members required for a system to still remain online and functioning.

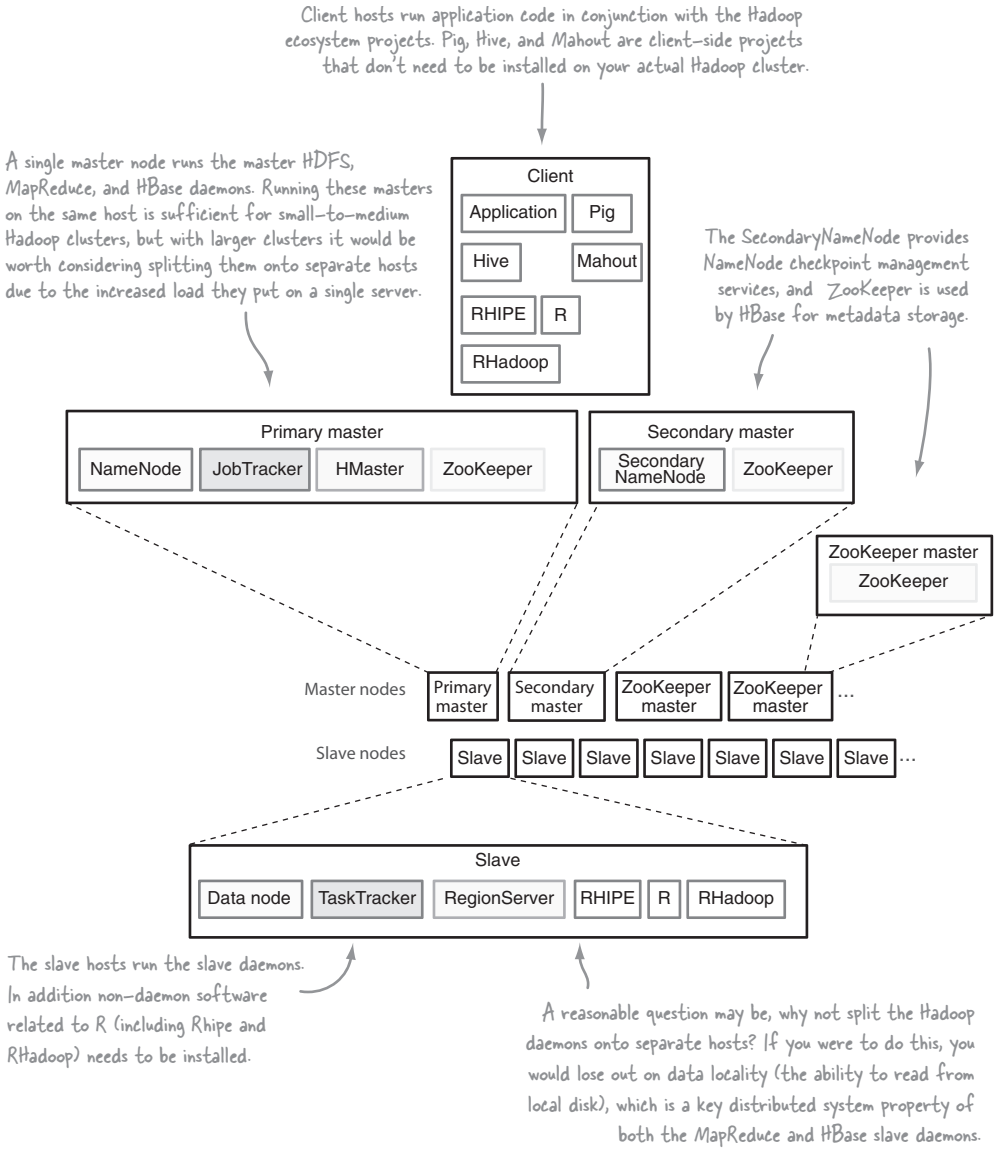


Figure 1.10 Hadoop's physical architecture

From a network topology perspective with regards to switches and firewalls, all of the master and slave nodes must be able to open connections to each other. For small clusters, all the hosts would run 1 GB network cards connected to a single, good-quality switch. For larger clusters look at 10 GB top-of-rack switches that have at least multiple 1 GB uplinks to dual-central switches. Client nodes also need to be able to talk to all of the master and slave nodes, but if necessary that access can be from behind a firewall that permits connection establishment only from the client side.

After reviewing Hadoop's physical architecture you've likely developed a good idea of who might benefit from using Hadoop. Let's take a look at companies currently using Hadoop, and in what capacity they're using it.

1.1.4 Who's using Hadoop?

Hadoop has a high level of penetration in high-tech companies, and is starting to make inroads across a broad range of sectors, including the enterprise (Booz Allen Hamilton, J.P. Morgan), government (NSA), and health care.

Facebook uses Hadoop, Hive, and HBase for data warehousing and real-time application serving.[8] Their data warehousing clusters are petabytes in size with thousands of nodes, and they use separate HBase-driven, real-time clusters for messaging and real-time analytics.

Twitter uses Hadoop, Pig, and HBase for data analysis, visualization, social graph analysis, and machine learning. Twitter LZO-compresses all of its data, and uses Protocol Buffers for serialization purposes, all of which are geared to optimizing the use of its storage and computing resources.

Yahoo! uses Hadoop for data analytics, machine learning, search ranking, email antispam, ad optimization, ETL,[9] and more. Combined, it has over 40,000 servers running Hadoop with 170 PB of storage.

eBay, Samsung, Rackspace, J.P. Morgan, Groupon, LinkedIn, AOL, Last.fm, and StumbleUpon are some of the other organizations that are also heavily invested in Hadoop. Microsoft is also starting to work with Hortonworks to ensure that Hadoop works on its platform.

Google, in its MapReduce paper, indicated that it used its version of MapReduce to create its web index from crawl data.[10] Google also highlights applications of MapReduce to include activities such as a distributed grep, URL access frequency (from log data), and a term-vector algorithm, which determines popular keywords for a host.

The organizations that use Hadoop grow by the day, and if you work at a Fortune 500 company you almost certainly use a Hadoop cluster in some capacity. It's clear that as Hadoop continues to mature, its adoption will continue to grow.

As with all technologies, a key part to being able to work effectively with Hadoop is to understand its shortcomings and design and architect your solutions to mitigate these as much as possible.

1.1.5 Hadoop limitations

Common areas identified as weaknesses across HDFS and MapReduce include availability and security. All of their master processes are single points of failure, although

---

[8] See http://www.facebook.com/note.php?note_id=468211193919.

[9] Extract, transform, and load (ETL) is the process by which data is extracted from outside sources, transformed to fit the project's needs, and loaded into the target data sink. ETL is a common process in data warehousing.

[10] In 2010 Google moved to a real-time indexing system called Caffeine: http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html.

you should note that there's active work on High Availability versions in the community. Security is another area that has its wrinkles, and again another area that's receiving focus.

HIGH AVAILABILITY

Until the Hadoop 2.x release, HDFS and MapReduce employed single-master models, resulting in single points of failure.[11] The Hadoop 2.x version will eventually bring both NameNode and JobTracker High Availability (HA) support. The 2.x NameNode HA design requires shared storage for NameNode metadata, which may require expensive HA storage. It supports a single standby NameNode, preferably on a separate rack.

SECURITY

Hadoop does offer a security model, but by default it's disabled. With the security model disabled, the only security feature that exists in Hadoop is HDFS file and directory-level ownership and permissions. But it's easy for malicious users to subvert and assume other users' identities. By default, all other Hadoop services are wide open, allowing any user to perform any kind of operation, such as killing another user's MapReduce jobs.

Hadoop can be configured to run with Kerberos, a network authentication protocol, which requires Hadoop daemons to authenticate clients, both user and other Hadoop components. Kerberos can be integrated with an organization's existing Active Directory, and therefore offers a single sign-on experience for users. Finally, and most important for the government sector, there's no storage or wire-level encryption in Hadoop. Overall, configuring Hadoop to be secure has a high pain point due to its complexity.

Let's examine the limitations of some of the individual systems.

HDFS

The weakness of HDFS is mainly around its lack of High Availability, its inefficient handling of small files, and its lack of transparent compression. HDFS isn't designed to work well with random reads over small files due to its optimization for sustained throughput. The community is waiting for append support for files, a feature that's nearing production readiness.

MAPREDUCE

MapReduce is a batch-based architecture, which means it doesn't lend itself to use cases that need real-time data access. Tasks that require global synchronization or sharing of mutable data aren't a good fit for MapReduce, because it's a shared-nothing architecture, which can pose challenges for some algorithms.

ECOSYSTEM VERSION COMPATIBILITIES

There also can be version-dependency challenges to running Hadoop. For example, HBase only works with a version of Hadoop that's not verified as production ready, due to its HDFS sync requirements (*sync* is a mechanism that ensures that all writes to a stream have been written to disk across all replicas). Hadoop versions 0.20.205 and

[11] In reality, the HDFS single point of failure may not be terribly significant; see http://goo.gl/1iSab.

newer, including 1.x and 2.x, include sync support, which will work with HBase. Other challenges with Hive and Hadoop also exist, where Hive may need to be recompiled to work with versions of Hadoop other than the one it was built against. Pig has had compatibility issues, too. For example, the Pig 0.8 release didn't work with Hadoop 0.20.203, requiring manual intervention to make them work together. This is one of the advantages to using a Hadoop distribution other than Apache, as these compatibility problems have been fixed.

One development worth tracking is the creation of BigTop (http://incubator.apache.org/projects/bigtop.html), currently an Apache incubator project, which is a contribution from Cloudera to open source its automated build and compliance system. It includes all of the major Hadoop ecosystem components and runs a number of integration tests to ensure they all work in conjunction with each other.

After tackling Hadoop's architecture and its weaknesses you're probably ready to roll up your sleeves and get hands-on with Hadoop, so let's take a look at how to get the Cloudera Distribution for Hadoop (CDH)[12] up and running on your system, which you can use for all the examples in this book.

1.2 Running Hadoop

The goal of this section is to show you how to run a MapReduce job on your host. To get there you'll need to install Cloudera's Hadoop distribution, run through some command-line and configuration steps, and write some MapReduce code.

1.2.1 Downloading and installing Hadoop

Cloudera includes the Cloudera Manager, a full-blown service and configuration management tool that works well for provisioning Hadoop clusters with multiple nodes. For this section we're interested in installing Hadoop on a single host, so we'll look at the individual packages that Cloudera offers. CDH includes OS-native installation packages for top-level Linux distributions such as RedHat, Debian, and SUSE, and their derivatives. Preinstalled CDH also includes tarball and Virtual Machine images. You can view all of the available options at http://www.cloudera.com/hadoop/.

Let's look at the instructions for installation on a RedHat-based Linux system (in this case you'll use CentOS). Appendix A includes the installation instructions for both the CDH tarball and the Apache Hadoop tarball.

RedHat uses packages called *RPMs* for installation, and Yum as a package installer that can fetch RPMs from remote Yum repositories. Cloudera hosts its own Yum repository containing Hadoop RPMs, which you'll use for installation.

You'll follow the pseudo-distributed installation instructions.[13] A pseudo-distributed setup is one where all of the Hadoop components are running on a single host. The first thing you need to do is download and install the "bootstrap" RPM, which will update your local Yum configuration to include Cloudera's remote Yum repository:

[12] I chose CDH for this task because of its simple installation and operation.

[13] See https://ccp.cloudera.com/display/CDHDOC/Installing+CDH3+on+a+Single+Linux+Node+in+Pseudo-distributed+Mode.

You need to run the wget and rpm commands as root.

```
$ sudo -s
$ wget http://archive.cloudera.com/redhat/cdh/cdh3-repository-1.0-1.noarch.rpm
$ rpm -ivh cdh3-repository-1.0-1.noarch.rpm
```

Next, you'll import Cloudera's RPM signing key so that Yum can verify the integrity of the RPMs that it downloads:

Note that we had to split this command across two lines, so you use the "\" character to escape the newline.

```
$ rpm --import \
  http://archive.cloudera.com/redhat/cdh/RPM-GPG-KEY-cloudera
```

The last step is to install the pseudo-distributed RPM package, which has dependencies on all the other core Hadoop RPMs. You'll also install Pig, Hive, and Snappy (which is contained in the Hadoop native package), because you'll be using them in this book:

```
$ yum install hadoop-0.20-conf-pseudo hadoop-0.20-native \
    hadoop-pig hadoop-hive
```

You've completed your installation of Hadoop. For this book you'll also be working with Oozie, HBase, and other projects, but you'll find instructions for these technologies in their respective sections.

JAVA VERSIONS

Hadoop requires version 1.6 update 8, or newer, of the Oracle Java Development Kit (JDK) on the host, which you can download from the Java SE Downloads (http://www.oracle.com/technetwork/java/javase/downloads/index.html) page.

You've installed the basics—it's time to learn how to configure Hadoop. Let's go over some basic commands so you can start and stop your cluster.

1.2.2 Hadoop configuration

After you've completed the installation instructions in the previous section, your software is ready for use without editing any configuration files. Knowing the basics of Hadoop's configuration is useful, so let's briefly touch upon it here. In CDH the Hadoop configs are contained under /etc/hadoop/conf. You'll find separate configuration files for different Hadoop components, and it's worth providing a quick overview of them in table 1.1.

Table 1.1 Hadoop configuration files

Filename	Description
hadoop-env.sh	Environment-specific settings go here. If a current JDK isn't in the system path you'll want to come here to configure your JAVA_HOME. You can also specify JVM options for various Hadoop components here. Customizing directory locations such as the log directory and the locations of the master and slave files is also performed here, although by default you shouldn't have to do any of what was just described in a CDH setup.
core-site.xml	Contains system-level Hadoop configuration items, such as the HDFS URL, the Hadoop temporary directory, and script locations for rack-aware Hadoop clusters. Settings in this file override the settings in core-default.xml. The default settings can be seen at http://hadoop.apache.org/common/docs/r1.0.0/core-default.html.
hdfs-site.xml	Contains HDFS settings such as the default file replication count, the block size, and whether permissions are enforced. To view the default settings you can look at http://hadoop.apache.org/common/docs/r1.0.0/hdfs-default.html. Settings in this file override the settings in hdfs-default.xml.
mapred-site.xml	HDFS settings such as the default number of reduce tasks, default min/max task memory sizes, and speculative execution are all set here. To view the default settings you can look at http://hadoop.apache.org/common/docs/r1.0.0/mapred-default.html. Settings in this file override the settings in mapred-default.xml.
masters	Contains a list of hosts that are Hadoop *masters*. This name is misleading and should have been called *secondary-masters*. When you start Hadoop it'll launch NameNode and JobTracker on the local host from which you issued the start command, and then SSH to all the nodes in this file to launch the SecondaryNameNode.
slaves	Contains a list of hosts that are Hadoop slaves. When you start Hadoop it will SSH to each host in this file and launch the DataNode and TaskTracker daemons.

The *site* XML files (those with *site* in their filenames) will grow as you start customizing your Hadoop cluster, and it can quickly become challenging to keep track of what changes you've made, and how they relate to the default configuration values. To help with this the author has written some code[14] that will compare the default and site files and indicate what properties have changed, as well as let you know about properties you may have misspelled. Some example output of the utility is included in the following code, which shows a few of the differences between the CDH core-default.xml and the core-site.xml files:

```
core-default.xml
Site file:  core-site.xml
                        Name   Final   Default                 Site
             fs.default.name   false  file:///  dfs://localhost:8020
    fs.har.impl.disable.cache   false      true                 null
hadoop.proxyuser.oozie.groups       -         -                    *
```

[14] https://github.com/alexholmes/hadoop-book/blob/master/src/main/java/com/manning/hip/ch1/ConfigDumper.java.

The Cloudera team has researched[15] more advanced techniques using static and dynamic methods to determine what options are supported in Hadoop, as well as discrepancies between application and Hadoop configurations.

1.2.3 Basic CLI commands

Let's rattle through the essentials you need to get up and running. First, start your cluster. You'll need sudo access for your user to run this command (it launches the Hadoop services via init.d scripts):

```
$ for svc in /etc/init.d/hadoop-0.20-*; do sudo $svc start; done
```

All the daemon log files are written under /var/log/hadoop. For example, the NameNode file is written to hadoop-hadoop-namenode-<HOSTNAME>.log, and this can be a useful file to look at if you have problems bringing up HDFS. You can test that things are up and running in a couple of ways. First try issuing a command to list the files in the root directory in HDFS:

```
$ hadoop fs -ls /
```

PATHNAME EXPANSIONS

You wouldn't think that the simple Hadoop filesystem command to list directory contents would have a quirk, but it does, and it's one that has bitten many a user, including the author, on numerous occasions. In bash and other shells it's normal to affix the * wildcard to filesystem commands, and for the shell to expand that prior to running a program. You would therefore (incorrectly) assume that the command `hadoop fs -ls /tmp/*` would work. But if you run this, and /tmp exists in your filesystem, your shell will expand the path based on the contents of /tmp on your local filesystem, and pass these filenames into Hadoop. At this point Hadoop will attempt to list files in HDFS that reside on your local system. The workaround is to prevent path expansion from occurring by enclosing the path in double quotes—this would become hadoop `fs -ls "/tmp/*"`.

If this works, HDFS is up and running. To make sure MapReduce is up and running you'll need to run a quick command to see what jobs are running:

```
$ hadoop job -list
0 jobs currently running
JobId    State    StartTime    UserName    Priority    SchedulingInfo
```

[15] See http://www.cloudera.com/blog/2011/08/automatically-documenting-apache-hadoop-configuration/.

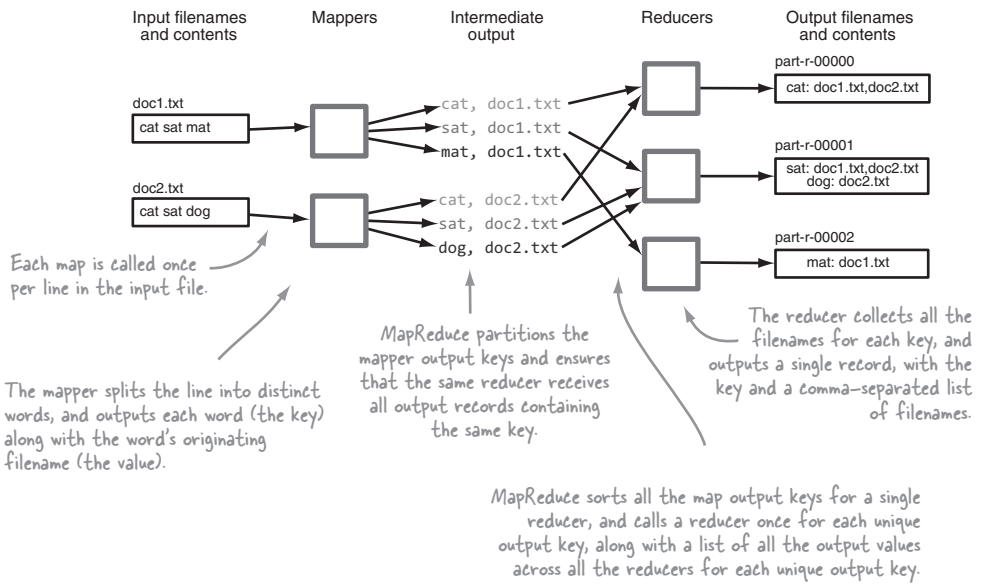


Figure 1.11 An example of an inverted index being created in MapReduce

Good, things seem to be in order. If you're curious about what commands you can issue from the command line take a look at http://hadoop.apache.org/common/docs/r1.0.0/file_system_shell.html for HDFS commands and http://hadoop.apache.org/common/docs/r1.0.0/commands_manual.html#job for MapReduce job commands.

Finally, to stop your cluster the process is similar to how you start it:

```
$ for svc in /etc/init.d/hadoop-0.20-*; do sudo $svc stop; done
```

With these essentials under your belt your next step is to write a MapReduce job (don't worry, it's not word count) that you can run in your new cluster.

1.2.4 Running a MapReduce job

Let's say you want to build an inverted index. MapReduce would be a good choice for this task because it can work on the creation of indexes in parallel, and as a result is a common MapReduce use case. Your input is a number of text files, and your output is a list of tuples, where each tuple is a word and a list of files that contain the word. Using standard processing techniques this would require you to find a mechanism to join all the words together. A naïve approach would be to perform this join in memory, but you may run out of memory if you have large numbers of unique keys. You could use an intermediary datastore such as a database, but that would be inefficient.

A better approach would be to tokenize each line and produce an intermediary file containing a word per line. Each of these intermediary files can then be sorted.

The final step would be to open all the sorted intermediary files and call a function for each unique word. This is what MapReduce does, albeit in a distributed fashion.

Figure 1.11 walks you through an example of a simple inverted index in MapReduce. Let's start by defining your mapper. Your reducers need to be able to generate a line for each word in your input, so your map output key should be each word in the input files so that MapReduce can join them all together. The value for each key will be the containing filename, which is your document ID. The following shows the mapper code:

When you extend the MapReduce mapper class you specify the key/value types for your inputs and outputs. You use the MapReduce default InputFormat for your job, which supplies keys as byte offsets into the input file, and values as each line in the file. Your map emits Text key/value pairs.

```
public static class Map
    extends Mapper<LongWritable, Text, Text, Text> {

  private Text documentId;

  private Text word = new Text();

  @Override
  protected void setup(Context context) {
    String filename =
        ((FileSplit) context.getInputSplit()).getPath().getName();
    documentId = new Text(filename);
  }

  @Override
  protected void map(LongWritable key, Text value,
                     Context context)
      throws IOException, InterruptedException {
    for (String token :
            StringUtils.split(value.toString())) {
      word.set(token);
      context.write(word, documentId);
    }
  }
}
```

A Text object to store the document ID (filename) for your input.

To cut down on object creation you create a single Text object, which you'll reuse.

Extract the filename from the context.

This method is called once at the start of the map and prior to the map method being called. You'll use this opportunity to store the input filename for this map.

This map method is called once per input line; map tasks are run in parallel over subsets of the input files.

For each word your map outputs the word as the key and the document ID as the value.

Your value contains an entire line from your file. You tokenize the line using StringUtils (which is far faster than using String.split).

The goal of your reducer is to create an output line for each word, and a list of the document IDs in which the word appears. The MapReduce framework will take care of calling your reducer once per unique key outputted by the mappers, along with a list of document IDs. All you need to do in your reducer is combine all the document IDs together and output them once in the reducer, as you can see in the next code block.

Much like your Map class you need to specify both the input and output key/value classes when you define your reducer.

```
public static class Reduce
    extends Reducer<Text, Text, Text, Text> {
  private Text docIds = new Text();
  public void reduce(Text key, Iterable<Text> values,
                     Context context)
      throws IOException, InterruptedException {

    HashSet<Text> uniqueDocIds = new HashSet<Text>();

    for (Text docId : values) {

      uniqueDocIds.add(new Text(docId));
    }

    docIds.set(new Text(StringUtils.join(uniqueDocIds, ",")));

    context.write(key, docIds);
  }
}
```

The reduce method is called once per unique map output key. The Iterable allows you to iterate over all the values that were emitted for the given key.

Keep a set of all the document IDs that you encounter for the key.

Iterate over all the DocumentIDs for the key.

Add the document ID to your set. The reason you create a new Text object is that MapReduce reuses the Text object when iterating over the values, which means you want to create a new copy.

Your reduce outputs the word, and a CSV-separated list of document IDs that contained the word.

The last step is to write the driver code that will set all the necessary properties to configure your MapReduce job to run. You need to let the framework know what classes should be used for the map and reduce functions, and also let it know where your input and output data is located. By default MapReduce assumes you're working with text; if you were working with more complex text structures, or altogether different data storage technologies, you would need to tell MapReduce how it should read and write from these data sources and sinks. The following source shows the full driver code:

Your input is 1 or more files, so create a sub-array from your input arguments, excluding the last item of the array, which is the MapReduce job output directory.

```
public static void main(String... args) throws Exception {

  runJob(
      Arrays.copyOfRange(args, 0, args.length - 1),
      args[args.length - 1]);
}

public static void runJob(String[] input, String output)
    throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf);

  job.setJarByClass(InvertedIndexMapReduce.class);

  job.setMapperClass(Map.class);

  job.setReducerClass(Reduce.class);

  job.setMapOutputKeyClass(Text.class);
  job.setMapOutputValueClass(Text.class);

  Path outputPath = new Path(output);
```

The Configuration container for your job configs. Anything that's set here is available to your map and reduce classes.

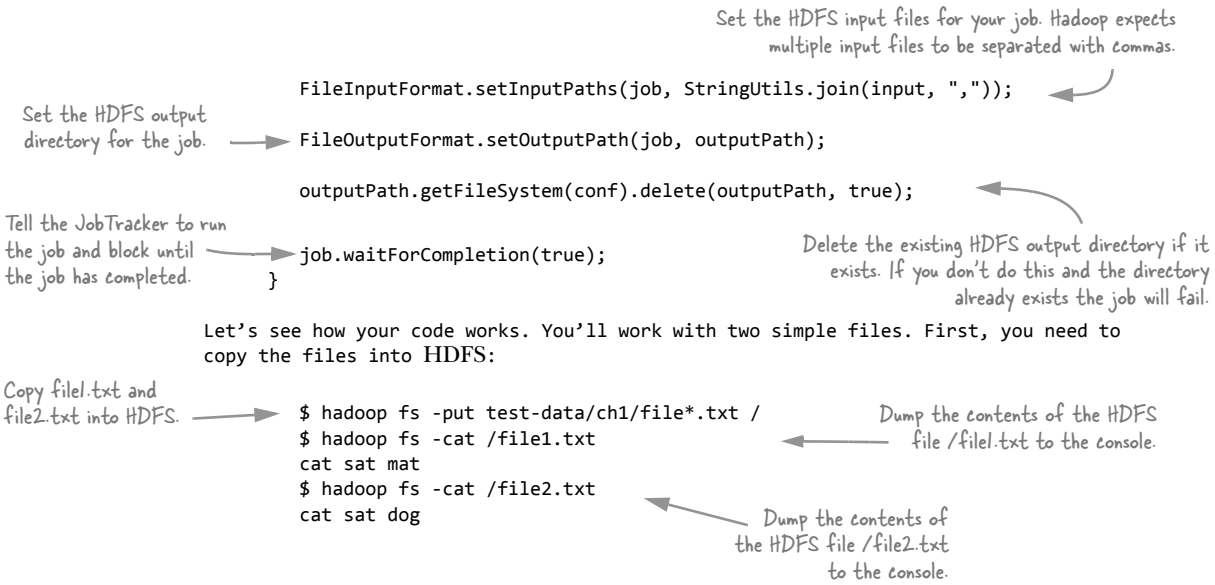
The Job class setJarByClass method determines the JAR that contains the class that's passed-in, which beneath the scenes is copied by Hadoop into the cluster and subsequently set in the Task's classpath so that your Map/Reduce classes are available to the Task.

Set the Map class that should be used for the job.

Set the Reduce class that should be used for the job.

If the map output key/value types differ from the input types you must tell Hadoop what they are. In this case your map will output each word and file as the key/value pairs, and both are Text objects.

Set the map output value class.



Next, run your MapReduce code. You'll use a shell script to run it, supplying the two input files as arguments, along with the job output directory:

```
$ export JAVA_HOME=<path to your JDK bin directory>
$ bin/run.sh com.manning.hip.ch1.InvertedIndexMapReduce \
  /file1.txt /file2.txt output
```

When your job completes you can examine HDFS for the job output files, and also view their contents:

```
$ hadoop fs -ls output/
Found 3 items
output/_SUCCESS
output/_logs
output/part-r-00000

$ hadoop fs -cat output/part-r-00000
cat    file2.txt,file1.txt
dog    file2.txt
mat    file1.txt
sat    file2.txt,file1.txt
```

You may be curious about where the map and reduce log files go. For that you need to know the job's ID, which will take you to the logs directory in the local filesystem. When you run your job from the command line, part of the output is the job ID, as follows:

```
...
INFO mapred.JobClient: Running job: job_201110271152_0001
...
```

With this ID in hand you can navigate to the directory on your local filesystem, which contains a directory for each map and reduce task. These tasks can be differentiated by the *m* and *r* in the directory names:

```
$ pwd
/var/log/hadoop-0.20/userlogs/job_201110271152_0001
$ ls -l
attempt_201110271152_0001_m_000000_0
attempt_201110271152_0001_m_000001_0
attempt_201110271152_0001_m_000002_0
attempt_201110271152_0001_m_000003_0
attempt_201110271152_0001_r_000000_0
```

Within each of the directories in the previous code there are three files, corresponding to standard out, standard error, and the system log (output from both the infrastructure task code, as well as any of your own log4j logging):

```
$ ls attempt_201110271152_0001_m_000000_0
stderr  stdout  syslog
```

Remember that in the pseudo-distributed setup everything's running on your local host, so it's easy to see everything in one place. On a true distributed cluster these logs

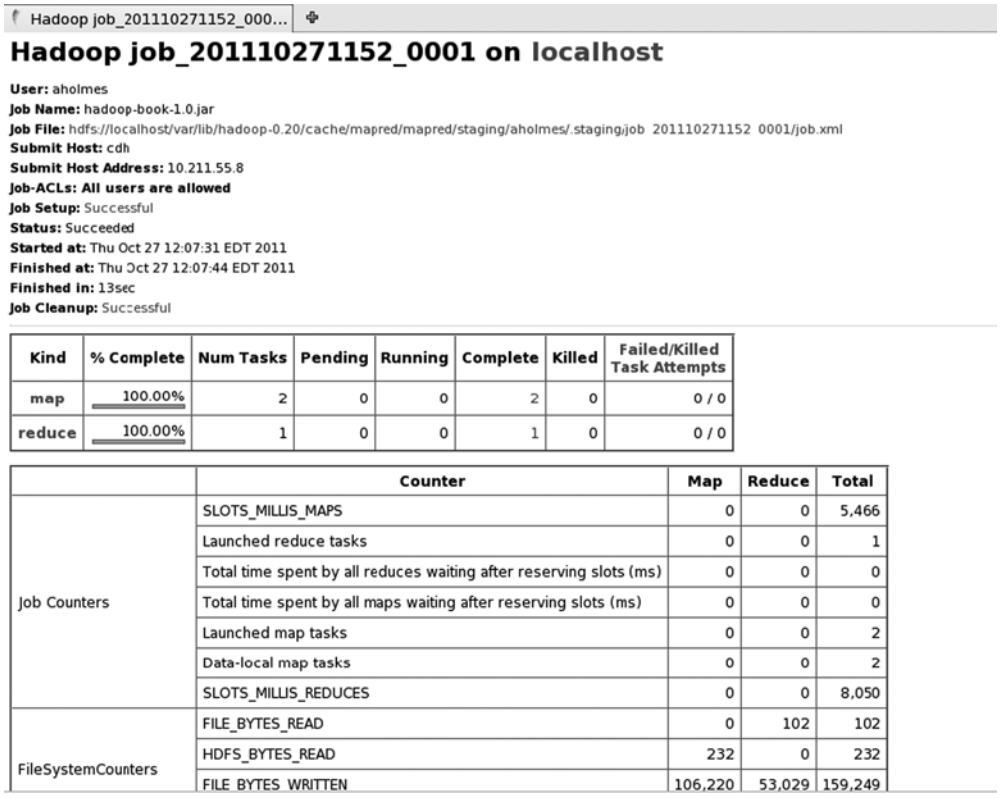


Figure 1.12 The Hadoop JobTracker user interface

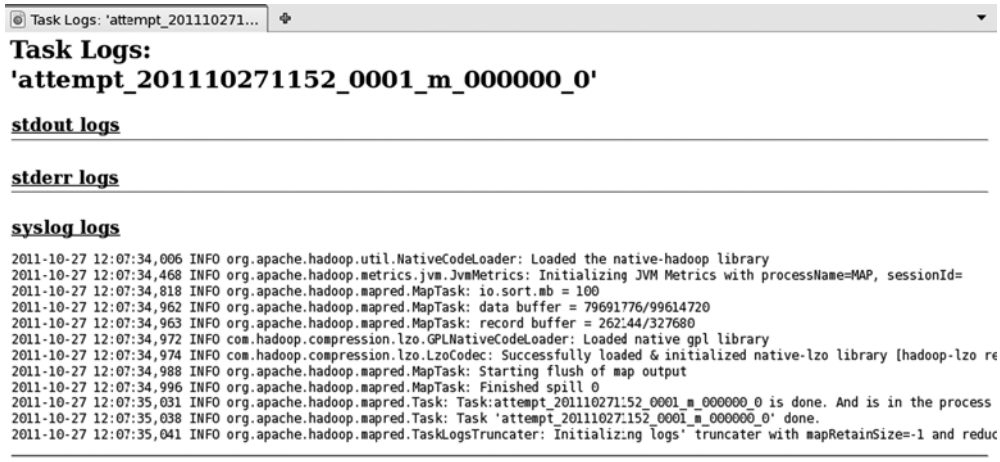


Figure 1.13 **The Hadoop TaskTracker user interface**

will be local to the remote TaskTracker nodes, which can make it harder to get to them. This is where the JobTracker and TaskTracker UI step in to provide easy access to the logs. Figures 1.12 and 1.13 show screenshots of the JobTracker summary page for your job, and the TaskTracker UI for one of the map tasks. In CDH you can access the JobTracker UI at http://localhost:50030/jobtracker.jsp.

This completes your whirlwind tour of how to run Hadoop.

1.3 Chapter summary

Hadoop is a distributed system designed to process, generate, and store large datasets. Its MapReduce implementation provides you with a fault-tolerant mechanism for large-scale data analysis. Hadoop also excels at working with heterogeneous structured and unstructured data sources at scale.

In this chapter, we examined Hadoop from functional and physical architectural standpoints. You also installed Hadoop and ran a MapReduce job.

The remainder of this book is dedicated to providing real-world techniques to solve common problems you encounter when working with Hadoop. You'll be introduced to a broad spectrum of subject areas, starting with HDFS and MapReduce, Pig, and Hive. You'll also look at data analysis techniques and explore technologies such as Mahout and Rhipe.

In chapter 2, the first stop on your journey, you'll discover how to bring data into (and out of) Hadoop. Without further ado, let's get started.