2 Starting Hadoop

This chapter covers

- The architectural components of Hadoop
- Setting up Hadoop and its three operating modes: standalone, pseudo-distributed, and fully distributed
- Web-based tools to monitor your Hadoop setup

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.

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
- JobTracker
- TaskTracker

We'll discuss each one and its role within Hadoop.

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 program to lower the workload on the machine. This means that the NameNode server doesn't double as a DataNode or a TaskTracker.

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

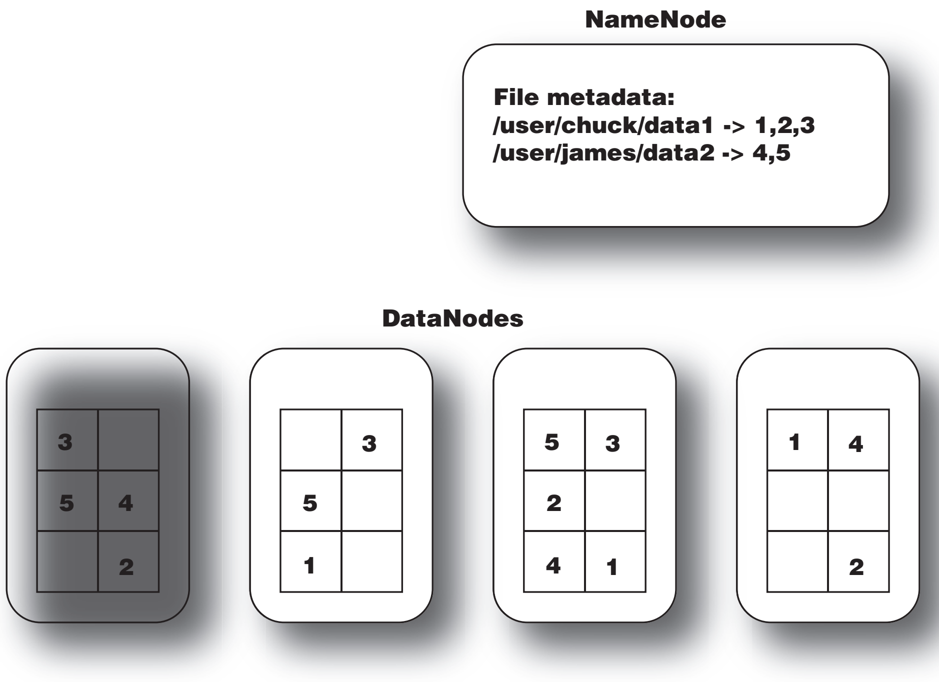


Figure 2.1 NameNode/DataNode interaction in HDFS. The NameNode keeps track of the file metadata—which files are in the system and how each file is broken down into blocks. The DataNodes provide backup store of the blocks and constantly report to the NameNode to keep the metadata current.

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.

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 or TaskTracker 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 *JobTracker*

The JobTracker daemon is the liaison between your application and Hadoop. Once you submit your code to your cluster, the JobTracker determines the execution plan by determining which files to process, assigns nodes to different tasks, and monitors all tasks as they're running. Should a task fail, the JobTracker will automatically relaunch the task, possibly on a different node, up to a predefined limit of retries.

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

2.1.5 *TaskTracker*

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

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

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

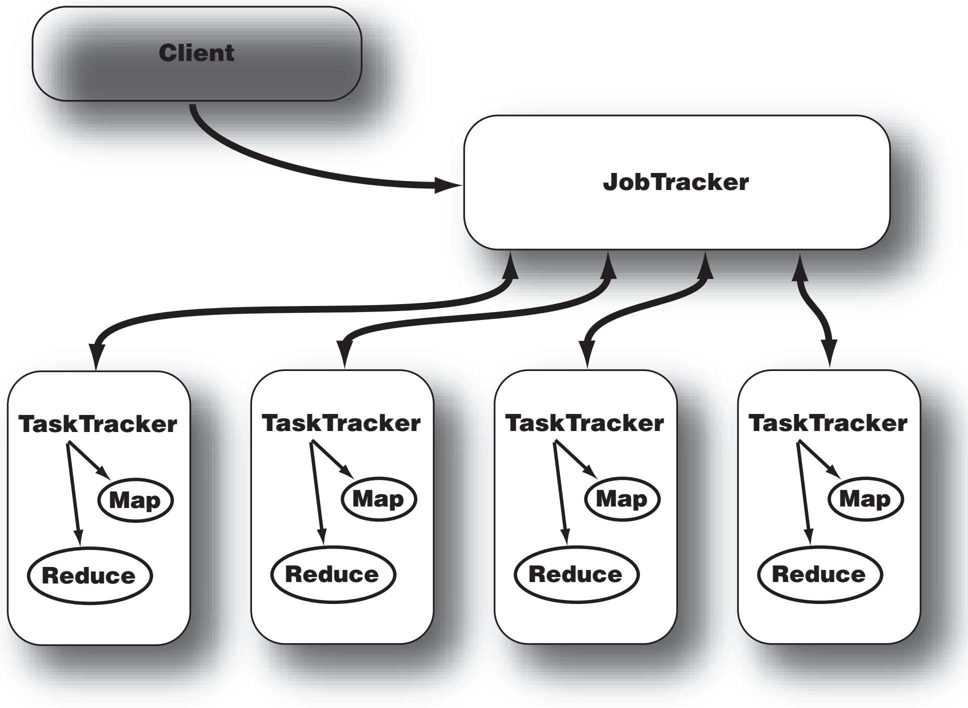


Figure 2.2 JobTracker and TaskTracker interaction. After a client calls the JobTracker to begin a data processing job, the JobTracker partitions the work and assigns different map and reduce tasks to each TaskTracker in the cluster.

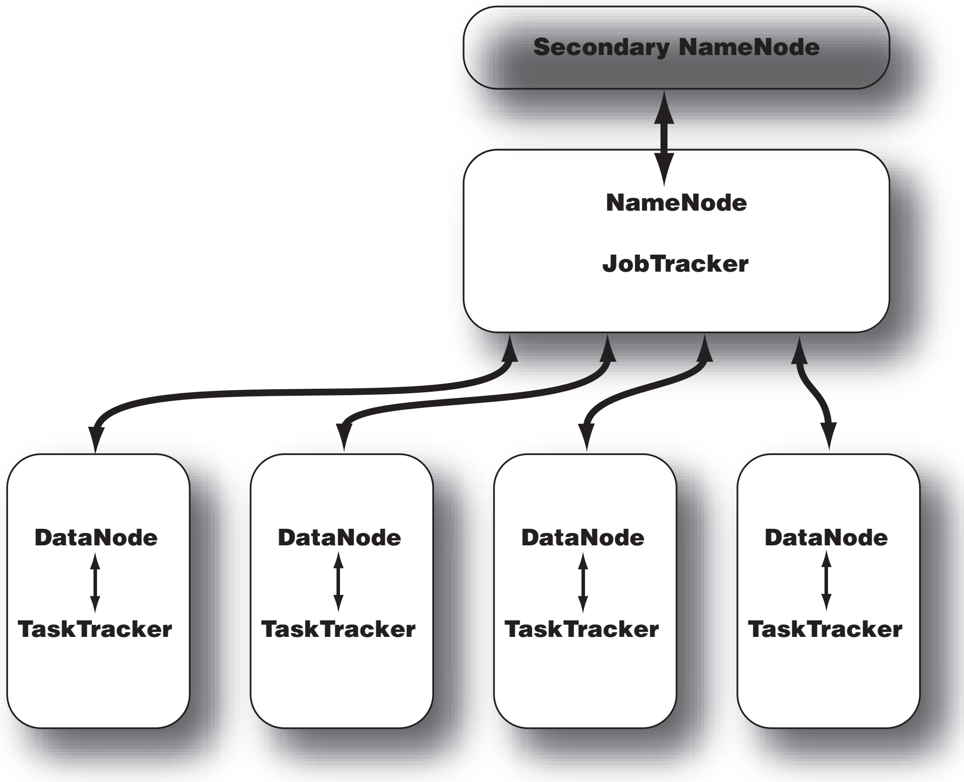


Figure 2.3 Topology of a typical Hadoop cluster. It's a master/slave architecture in which the NameNode and JobTracker are masters and the DataNodes and TaskTrackers are slaves.

Having covered each of the Hadoop daemons, we depict the topology of one typical Hadoop cluster in figure 2.3.

This topology features a master node running the NameNode and JobTracker daemons and a standalone node with the SNN in case the master node fails. For small clusters, the SNN can reside on one of the slave nodes. On the other hand, for large clusters, separate the NameNode and JobTracker on two machines. The slave machines each host a DataNode and TaskTracker, 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 the next section 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 2.3.

2.2 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.

2.2.1 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.

2.2.2 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.)

2.2.3 Generate SSH key pair

Having verified that SSH is correctly installed on all nodes of the cluster, we use `ssh-keygen` 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
```

and we next need to distribute this public key across your cluster.

2.2.4 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.

2.3 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 $HADOOP_HOME
[hadoop-user@master]$ ls -l conf/
total 100
```

```
-rw-rw-r-- 1 hadoop-user hadoop 2065 Dec 1 10:07 capacity-scheduler.xml
-rw-rw-r-- 1 hadoop-user hadoop  535 Dec 1 10:07 configuration.xsl
-rw-rw-r-- 1 hadoop-user hadoop 49456 Dec 1 10:07 hadoop-default.xml
-rwxrwxr-x 1 hadoop-user hadoop 2314 Jan 8 17:01 hadoop-env.sh
-rw-rw-r-- 1 hadoop-user hadoop 2234 Jan 2 15:29 hadoop-site.xml
-rw-rw-r-- 1 hadoop-user hadoop 2815 Dec 1 10:07 log4j.properties
-rw-rw-r-- 1 hadoop-user hadoop  28 Jan 2 15:29 masters
-rw-rw-r-- 1 hadoop-user hadoop  84 Jan 2 15:29 slaves
-rw-rw-r-- 1 hadoop-user hadoop  401 Dec 1 10:07 sslinfo.xml.example
```

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 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 file contains 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 are hadoop-default.xml and hadoop-site.xml. As the names imply, hadoop-default.xml contains the default Hadoop settings to be used unless they are explicitly overridden in hadoop-site.xml. In practice you only deal with hadoop-site.xml. In version 0.20 this file has been separated out into three XML files: core-site.xml, hdfs-site.xml, and mapred-site.xml. This refactoring better aligns the configuration settings to the subsystem of Hadoop that they control. In the rest of this chapter we'll generally point out which of the three files 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 hadoop-site.xml.

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

2.3.1 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 three XML files (or hadoop-site.xml before version 0.20) are empty under this default mode:

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

<!-- 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. Its 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.

2.3.2 Pseudo-distributed mode

The pseudo-distributed mode is running Hadoop in a "cluster of one" with all daemons running on a single machine. This mode complements the standalone mode for debugging your code, 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.

Listing 2.1 Example of the three 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.default.name</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>
```

mapred-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>mapred.job.tracker</name>
 <value>localhost:9001</value>
 <description>The host and port that the MapReduce job tracker runs
 at.</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 and mapred-site.xml we specify the hostname and port of the NameNode and the JobTracker, respectively. 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. We must also specify the location of the Secondary NameNode in the masters file and the slave nodes in the slaves file:

```
[hadoop-user@master]$ cat masters
localhost
[hadoop-user@master]$ cat slaves
localhost
```

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 daemons by use of the start-all.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
26893 Jps
26832 TaskTracker
26620 SecondaryNameNode
26333 NameNode
26484 DataNode
26703 JobTracker
```

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

```
[hadoop-user@master]$ bin/stop-all.sh
```

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.

2.3.3 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 JobTracker daemons
- *backup*—The server that hosts the Secondary NameNode daemon
- *hadoop1, hadoop2, hadoop3, ...*—The slave boxes of the cluster running both DataNode and TaskTracker daemons

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

Listing 2.2 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.default.name</name>
 <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>
```

(1) **Locate NameNode for filesystem**

mapred-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>mapred.job.tracker</name>
 <value>master:9001</value>
 <description>The host and port that the MapReduce job tracker runs
 at.</description>
</property>

</configuration>
```

2 **Locate JobTracker master**

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

</configuration>
```

3 **Increase HDFS replication factor**

The key differences are

- We explicitly stated the hostname for location of the NameNode (1) and JobTracker (2) daemons.
- We increased the HDFS replication factor to take advantage of distributed storage (3). 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.

```
[hadoop-user@master]$ cat masters
backup
[hadoop-user@master]$ cat slaves
hadoop1
hadoop2
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]$ bin/start-all.sh
```

and verify the nodes are running their assigned jobs.

```
[hadoop-user@master]$ jps
30879 JobTracker
30717 NameNode
```

```
30965 Jps
[hadoop-user@backup]$ jps
2099 Jps
1679 SecondaryNameNode
[hadoop-user@hadoop1]$ jps
7101 TaskTracker
7617 Jps
6988 DataNode
```

You have a functioning cluster!

Switching between modes

A practice that I 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.

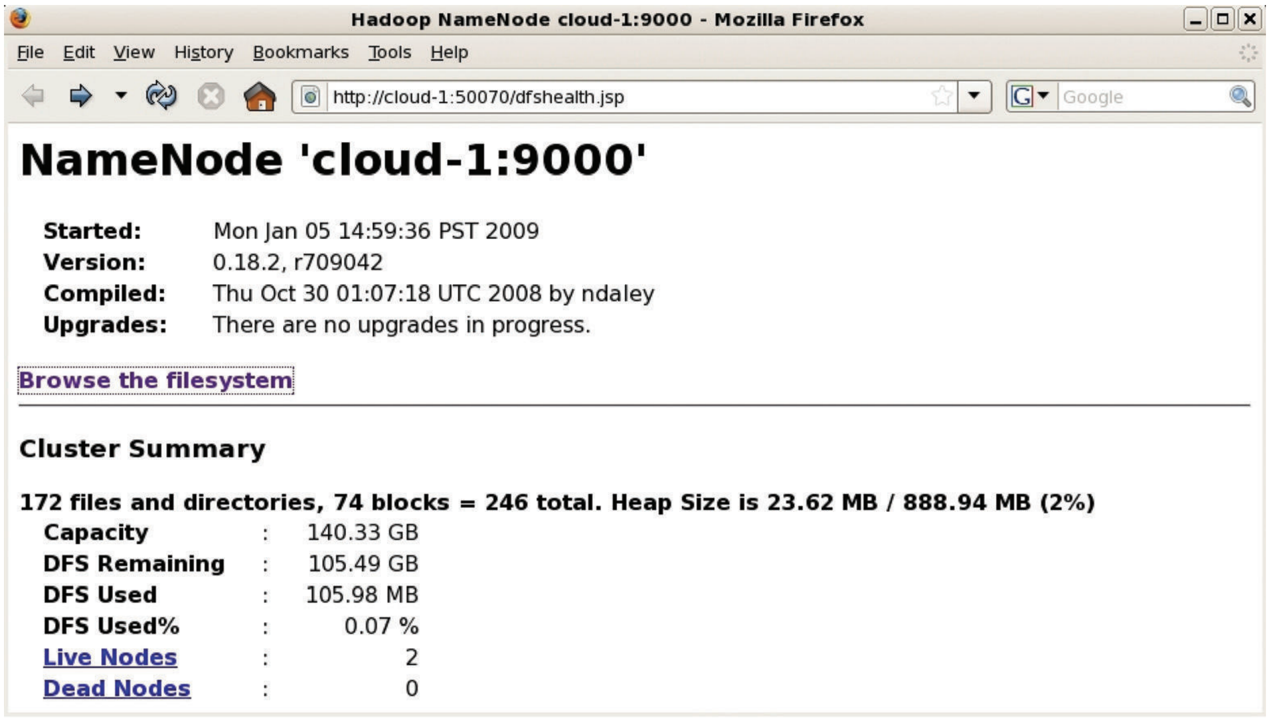
2.4 Web-based cluster UI

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 2-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 ongoing MapReduce jobs. Figure 2.5 depicts one hosted at port 50030 of the JobTracker.

Again, a wealth of information is available through this reporting interface. You can access the status of ongoing MapReduce tasks 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. Finally, the Hadoop configuration for each job is also available, as shown in figure 2.6. With all of this information you can streamline your MapReduce programs to better utilize the resources of your cluster.


Hadoop NameNode cloud-1:9000 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://cloud-1:50070/dfshealth.jsp
Google

NameNode 'cloud-1:9000'

Started:	Mon Jan 05 14:59:36 PST 2009
Version:	0.18.2, r709042
Compiled:	Thu Oct 30 01:07:18 UTC 2008 by ndaley
Upgrades:	There are no upgrades in progress.

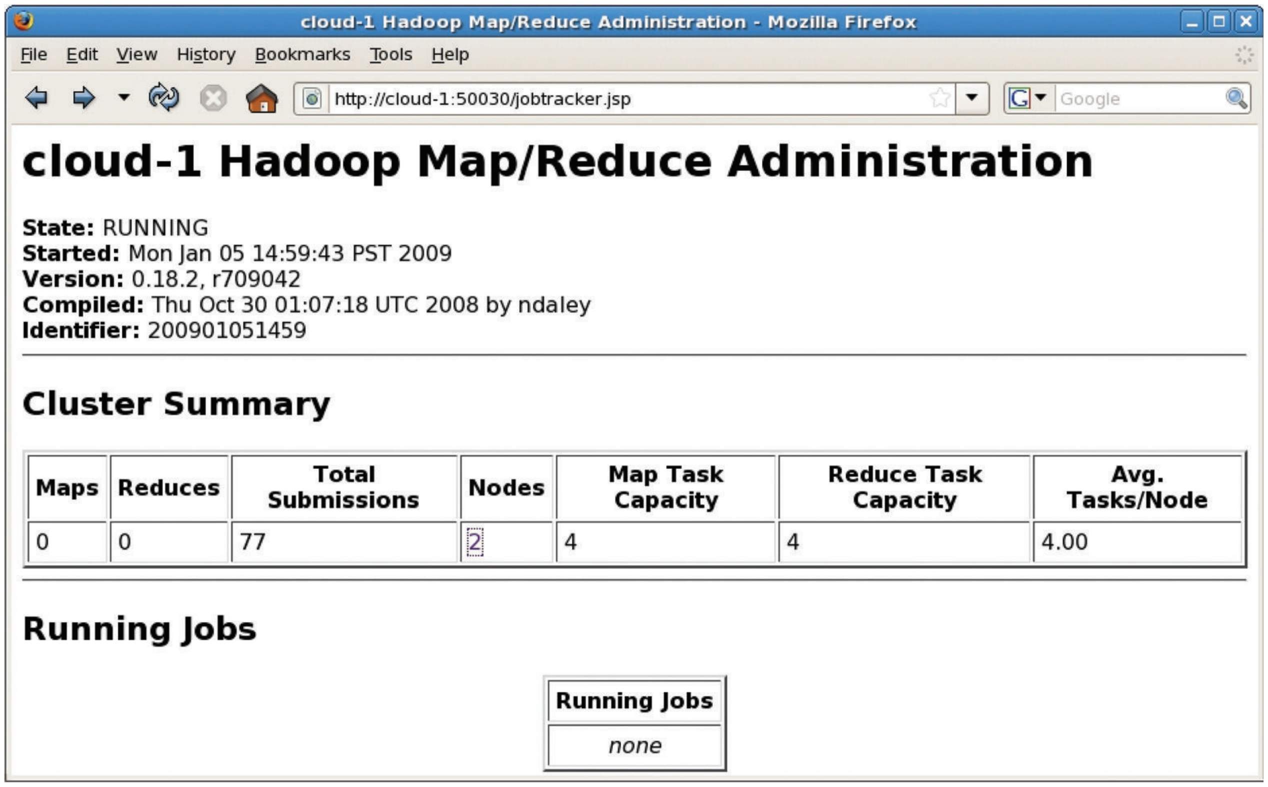
Browse the filesystem

Cluster Summary

172 files and directories, 74 blocks = 246 total. Heap Size is 23.62 MB / 888.94 MB (2%)

Capacity	:	140.33 GB
DFS Remaining	:	105.49 GB
DFS Used	:	105.98 MB
DFS Used%	:	0.07 %
Live Nodes	:	2
Dead 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.


cloud-1 Hadoop Map/Reduce Administration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://cloud-1:50030/jobtracker.jsp
Google

cloud-1 Hadoop Map/Reduce Administration

State: RUNNING
Started: Mon Jan 05 14:59:43 PST 2009
Version: 0.18.2, r709042
Compiled: Thu Oct 30 01:07:18 UTC 2008 by ndaley
Identifier: 200901051459

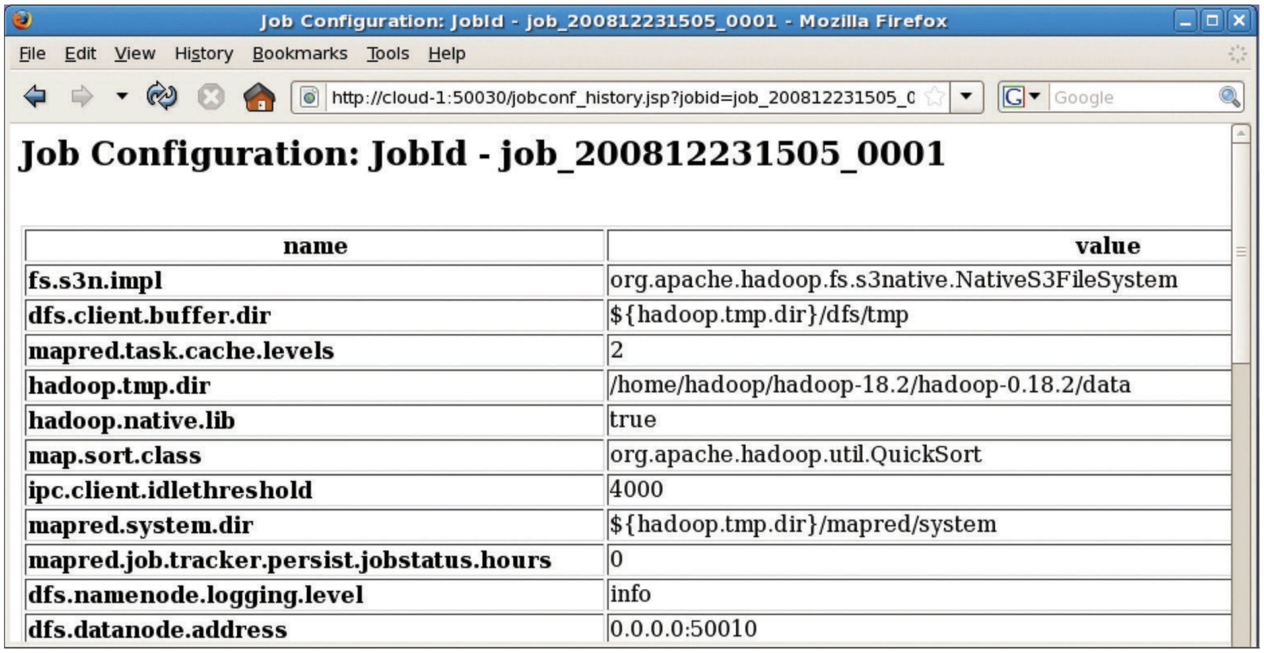
Cluster Summary

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node
0	0	77	2	4	4	4.00

Running Jobs

Running Jobs
none

Figure 2.5 A snapshot of the MapReduce web interface. This tool allows you to monitor active MapReduce jobs and access the logs of each map and reduce task. The logs of previously submitted jobs are also available and are useful for debugging your programs.


Job Configuration: JobId - job_200812231505_0001 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://cloud-1:50030/jobconf_history.jsp?jobid=job_200812231505_0
Google

Job Configuration: JobId - job_200812231505_0001

name	value
fs.s3n.impl	org.apache.hadoop.fs.s3native.NativeS3FileSystem
dfs.client.buffer.dir	${hadoop.tmp.dir}/dfs/tmp
mapred.task.cache.levels	2
hadoop.tmp.dir	/home/hadoop/hadoop-18.2/hadoop-0.18.2/data
hadoop.native.lib	true
map.sort.class	org.apache.hadoop.util.QuickSort
ipc.client.idlethreshold	4000
mapred.system.dir	${hadoop.tmp.dir}/mapred/system
mapred.job.tracker.persist.jobstatus.hours	0
dfs.namenode.logging.level	info
dfs.datanode.address	0.0.0.0:50010

Figure 2.6 Configuration details for a specific MapReduce job. This information is potentially useful when tuning parameters to optimize the performance of 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.

2.5 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.

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 hadoop-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 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.

3.1 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,[1] 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.

3.1.1 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[2]

```
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

[1] There are several ongoing projects that try to make HDFS mountable as a Unix filesystem. More details are at http://wiki.apache.org/hadoop/MountableHDFS. As of this writing these projects aren't officially part of Hadoop and they may not have the reliability needed for some production systems.
[2] 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.

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 9, 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 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/chuck will look like hdfs://localhost:9000/user/chuck/example.txt. You can use the Hadoop `cat` command to show the content of that file:

hadoop fs -cat hdfs://localhost:9000/user/chuck/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 conf/core-site.xml file to the pseudo-distributed configuration, your fs.default.name property in the file should be

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

Under this configuration, shorten the URI hdfs://localhost:9000/user/chuck/example.txt to /user/chuck/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/chuck/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 *chuck*. You should substitute your user name in the example commands.

```
hadoop fs -mkdir /user/chuck
```

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   - chuck supergroup          0 2009-01-14 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   - chuck supergroup          0 2009-01-14 10:23 /user
drwxr-xr-x   - chuck supergroup          0 2009-01-14 10:23 /user/chuck
```

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/chuck
```

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

```
$ hadoop fs -lsr /
drwxr-xr-x   - chuck supergroup          0 2009-01-14 10:23 /user
drwxr-xr-x   - chuck supergroup          0 2009-01-14 11:02 /user/chuck
-rw-r--r--   1 chuck supergroup        264 2009-01-14 11:02
➥/user/chuck/example.txt
```

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 chuck supergroup        264 2009-01-14 11:02
➥/user/chuck/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 <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
                        dirName (full path) <dir>
                and file entries are of the form
                        fileName(full path) <r n> size
                where n is the number of replicas specified for the file
                and size is the size of the file, in bytes.
```

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.

3.1.2 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 3.1.

Listing 3.1 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]);                        ❶ Specify input directory
        Path hdfsFile = new Path(args[1]);                          and output file

        try {                                                        Get list of
            FileStatus[] inputFiles = local.listStatus(inputDir); ❷ local files
            FSDataOutputStream out = hdfs.create(hdfsFile); ❸ Create
                                                              HDFS output stream
            for (int i=0; i<inputFiles.length; i++) {
                System.out.println(inputFiles[i].getPath().getName());
                FSDataInputStream in =
        ➥ local.open(inputFiles[i].getPath());                  ❹ Open local
                byte buffer[] = new byte[256];                      input stream
                int bytesRead = 0;
                while( (bytesRead = in.read(buffer)) > 0) {
                    out.write(buffer, 0, bytesRead);
                }
                in.close();
            }
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

The general flow of the program involves first setting the local directory and the HDFS destination file based on user-specified arguments ❶. In ❷ we extract information about each file in the local input directory. We create an output stream to write to the HDFS file in ❸. We loop through each file in the local directory, and ❹ 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.

3.2 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 3.1 displays a high-level diagram of the entire process, and we further dissect each component as we step through the flow.

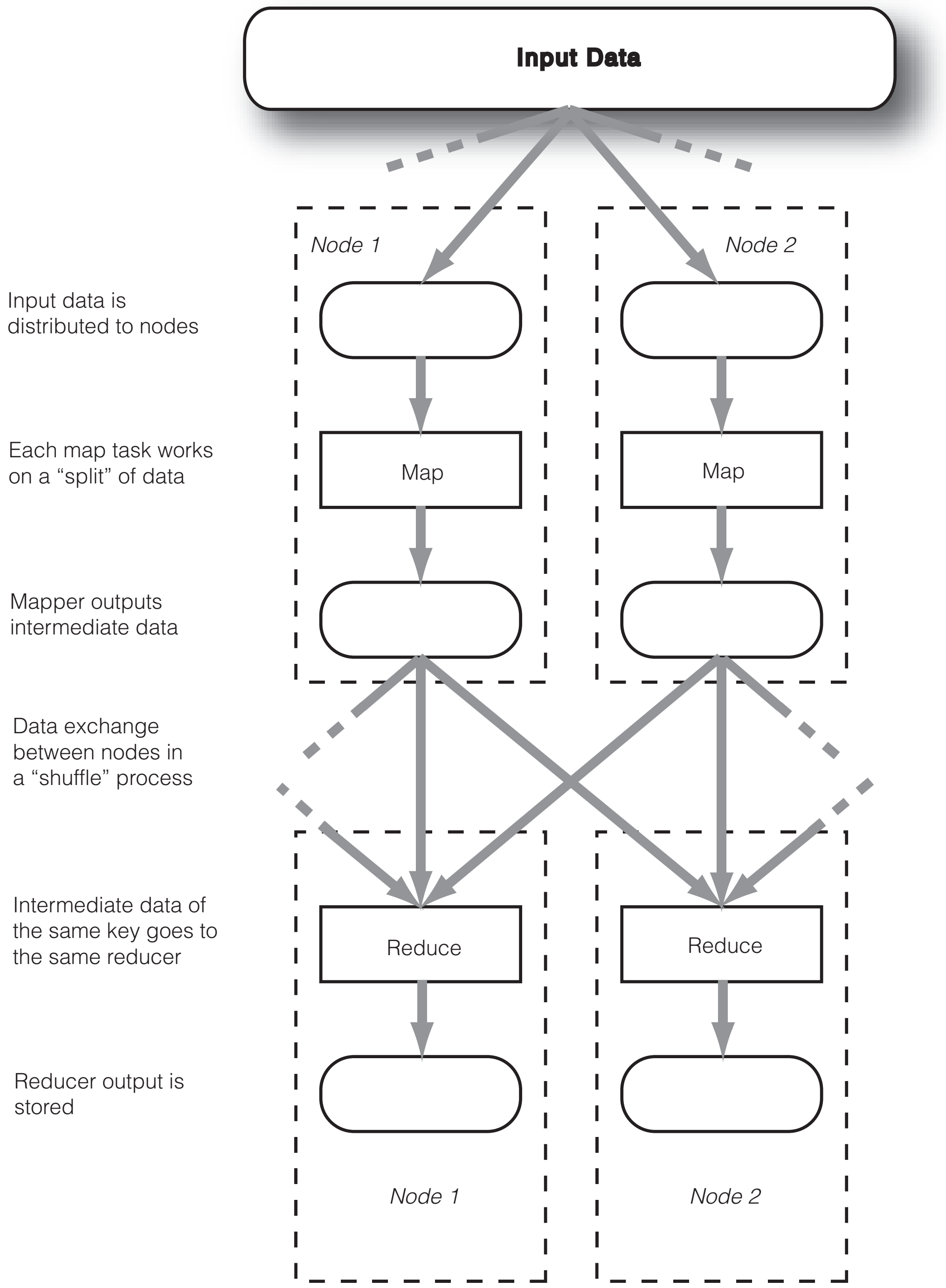

Input Data
Node 1
Node 2
Input data is distributed to nodes
Each map task works on a "split" of data
Map
Map
Mapper outputs intermediate data
Data exchange between nodes in a "shuffle" process
Intermediate data of the same key goes to the same reducer
Reduce
Reduce
Reducer output is stored
Node 1
Node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

Figure 3.1 The 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.

3.2.1 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 3.1.

Table 3.1 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
`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 which Hadoop natively supports. You can create your own custom type as long as it implements the `Writable` (or `WritableComparable<T>`) interface. For example, listing 3.2 shows a class that can represent edges in a network. This may represent a flight route between two cities.

Listing 3.2 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 Specify how to
        departureNode = in.readUTF();                                 read data in
        arrivalNode = in.readUTF();
    }

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

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

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 3.1: the mapper.

3.2.2 Mapper

To serve as the mapper, a class implements from the Mapper interface and inherits the `MapReduceBase` class. The `MapReduceBase` class, not surprisingly, serves as the base class for both mappers and reducers. It includes two methods that effectively act as the constructor and destructor for the class:

- *void configure(JobConf job)*—In this function you can extract the parameters set either by the configuration XML files or in the main class of your application. Call this function before any data processing begins.
- *void close()*—As the last action before the map task terminates, this function should wrap up any loose ends—database connections, open files, and so on.

The `Mapper` interface 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,
         OutputCollector<K2,V2> output,
         Reporter reporter
        ) throws IOException
```

The function generates a (possibly empty) list of (K2, V2) pairs for a given (K1, V1) input pair. The `OutputCollector` receives the output of the mapping process, and the `Reporter` provides 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 3.2.

Table 3.2 Some useful `Mapper` implementations predefined by Hadoop

Class	Description
`IdentityMapper<K,V>`	Implements Mapper<K,V,K,V> and maps inputs directly to outputs
`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
`TokenCountMapper<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 3.1.

3.2.3 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,
            OutputCollector<K3,V3> output,
            Reporter reporter
           ) 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 `OutputCollector` receives the output of the reduce process and writes it to an output file. The `Reporter` provides the option to record extra information about the reducer as the task progresses.

Table 3.3 lists a couple of basic reducer implementations provided by Hadoop.

Table 3.3 Some useful `Reducer` implementations predefined by Hadoop

Class	Description
`IdentityReducer<K,V>`	Implements Reducer<K,V,K,V> and maps inputs directly to outputs
`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.

3.2.4 *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)	James Warren
...	

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;
    }

    @Override
    public void configure(JobConf conf) { }
}
```

A custom partitioner only needs to implement two functions: `configure()` and `getPartition()`. The former uses the Hadoop job configuration to configure the partitioner, and the latter returns an integer between 0 and the number of reduce tasks indexing to which reducer the (key/value) pair will be sent.

The exact mechanics of the partitioner may be difficult to follow. Figure 3.2 illustrates this for better understanding.

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.

3.2.5 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

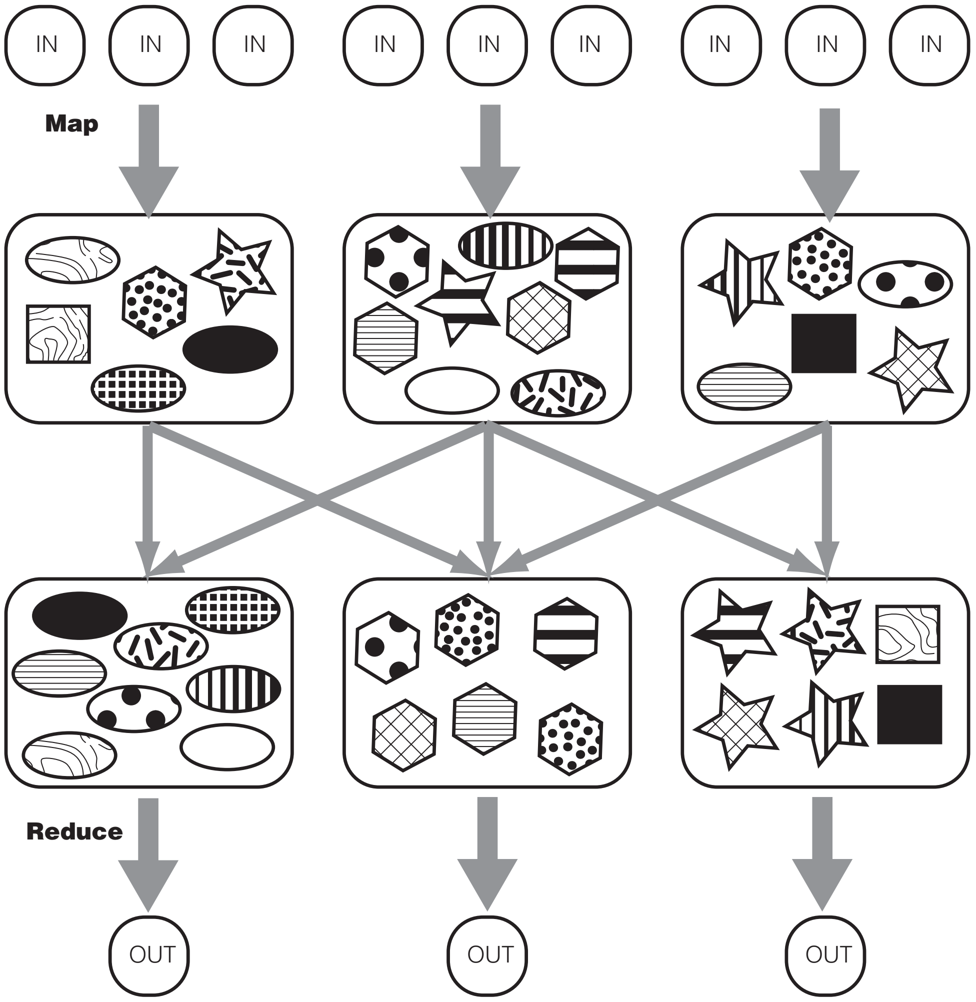


Figure 3.2 The MapReduce data flow, with an emphasis on partitioning and shuffling. Each icon is a key/value pair. The shapes represents keys, whereas the inner patterns represent values. After shuffling, all icons of the same shape (key) are in the same reducer. Different keys can go to the same reducer, as seen in the rightmost reducer. The partitioner decides which key goes where. Note that the leftmost reducer has more load due to more data under the "ellipse" key.

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.

3.2.6 *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 3.3), using some of the classes we've learned.

Listing 3.3 Revised version of the `WordCount` example

```
public class WordCount2 {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(WordCount2.class);

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

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(LongWritable.class);
        conf.setMapperClass(TokenCountMapper.class);        ❶ Hadoop's own TokenCountMapper
        conf.setCombinerClass(LongSumReducer.class);
        conf.setReducerClass(LongSumReducer.class);         ❷ Hadoop's own LongSumReducer

        client.setConf(conf);
        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

We have to write only the driver for this MapReduce program because we have used Hadoop's predefined `TokenCountMapper` class ❶ and `LongSumReducer` class ❷. 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.

3.3 *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.

3.3.1 *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 3.4 lists other popular implementations of `InputFormat` along with a description of the key/value pair each one passes to the mapper.

Table 3.4 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 `JobConf` object to use the `KeyValueTextInputFormat` class to read this file.

```
conf.setInputFormat(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> {
  InputSplit[] getSplits(JobConf job, int numSplits) throws IOException;
  RecordReader<K, V> getRecordReader(InputSplit split,
                                     JobConf job,
                                     Reporter reporter) throws IOException;
}
```

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 3.4 subclass `FileInputFormat`. `FileInputFormat` implements the `getSplits()` method but leaves `getRecordReader()` abstract for the subclass to fill out. `FileInputFormat`'s `getSplits()` implementation tries to divide the input data into roughly the number of splits specified in `numSplits`, subject to the constraints that each split must have more than `mapred.min.split.size` 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 next(K key, V value) throws IOException;

  K createKey();
  V createValue();

  long getPos() throws IOException;
  public void close() throws IOException;
  float getProgress() 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[3]. 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 `KeyValue-TextInputFormat`, 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
```

[3] 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.

3.3.2 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 `RecordReaders` parse the format of the input.

Hadoop provides several standard implementations of `OutputFormat,` as shown in table 3.5. Not surprisingly, almost all the ones we deal with inherit from the `File OutputFormat` 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 3.5 **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.

3.4 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.