APPENDIX A

Installing Apache Hadoop

It's easy to install Hadoop on a single machine to try it out. (For installation on a cluster, refer to Chapter 10.)

In this appendix, we cover how to install Hadoop Common, HDFS, MapReduce, and YARN using a binary tarball release from the Apache Software Foundation. Instructions for installing the other projects covered in this book are included at the start of the relevant chapters.

Another option is to use a virtual machine (such as Cloudera's Quick-Start VM) that comes with all the Hadoop services preinstalled and configured.

The instructions that follow are suitable for Unix-based systems, including Mac OS X (which is not a production platform, but is fine for development).

Prerequisites

Make sure you have a suitable version of Java installed. You can check the Hadoop wiki to find which version you need. The following command confirms that Java was installed correctly:

```
% java -version
java version "1.7.0_25"
Java(TM) SE Runtime Environment (build 1.7.0_25-b15)
Java HotSpot(TM) 64-Bit Server VM (build 23.25-b01, mixed mode)
```

Installation

Start by deciding which user you'd like to run Hadoop as. For trying out Hadoop or developing Hadoop programs, you can run Hadoop on a single machine using your own user account.

Download a stable release, which is packaged as a gzipped tar file, from the Apache Hadoop releases page, and unpack it somewhere on your filesystem:

```
% tar xzf hadoop-x.y.z.tar.gz
```

Before you can run Hadoop, you need to tell it where Java is located on your system. If you have the `JAVA_HOME` environment variable set to point to a suitable Java installation, that will be used, and you don't have to configure anything further. (It is often set in a shell startup file, such as *~/.bash_profile* or *~/.bashrc*.) Otherwise, you can set the Java installation that Hadoop uses by editing *conf/hadoop-env.sh* and specifying the `JAVA_HOME` variable. For example, on my Mac, I changed the line to read:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.7.0_25.jdk/Contents/Home
```

to point to the installed version of Java.

It's very convenient to create an environment variable that points to the Hadoop installation directory (`HADOOP_HOME`, by convention) and to put the Hadoop binary directories on your command-line path. For example:

```
% export HADOOP_HOME=~/sw/hadoop-x.y.z
% export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

Note that the *sbin* directory contains the scripts for running Hadoop daemons, so it should be included if you plan to run the daemons on your local machine.

Check that Hadoop runs by typing:

```
% hadoop version
Hadoop 2.5.1
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r 2e18d179e4a8065
b6a9f29cf2de9451891265cce
Compiled by jenkins on 2014-09-05T23:11Z
Compiled with protoc 2.5.0
From source with checksum 6424fcab95bfff8337780a181ad7c78
This command was run using /Users/tom/sw/hadoop-2.5.1/share/hadoop/common/hadoop
-common-2.5.1.jar
```

Configuration

Each component in Hadoop is configured using an XML file. Common properties go in *core-site.xml*, and properties pertaining to HDFS, MapReduce, and YARN go into the appropriately named file: *hdfs-site.xml*, *mapred-site.xml*, and *yarn-site.xml*. These files are all located in the *etc/hadoop* subdirectory.

You can see the default settings for all the properties that are governed by these configuration files by looking in the *share/doc* directory hierarchy of your Hadoop installation for files called *core-default.xml*, *hdfs-default.xml*, *mapred-default.xml*, and *yarn-default.xml*.

Hadoop can be run in one of three modes:

Standalone (or local) mode
: There are no daemons running and everything runs in a single JVM. Standalone mode is suitable for running MapReduce programs during development, since it is easy to test and debug them.

Pseudodistributed mode
: The Hadoop daemons run on the local machine, thus simulating a cluster on a small scale.

Fully distributed mode
: The Hadoop daemons run on a cluster of machines. This setup is described in Chapter 10.

To run Hadoop in a particular mode, you need to do two things: set the appropriate properties, and start the Hadoop daemons. Table A-1 shows the minimal set of properties to configure each mode. In standalone mode, the local filesystem and the local MapReduce job runner are used. In the distributed modes, the HDFS and YARN daemons are started, and MapReduce is configured to use YARN.

Table A-1. Key configuration properties for different modes

Component	Property	Standalone	Pseudodistributed	Fully distributed
Common	`fs.defaultFS`	`file:///` (default)	`hdfs://localhost/`	`hdfs://`*`namenode`*`/`
HDFS	`dfs.replication`	N/A	`1`	`3` (default)
MapReduce	`mapreduce.framework.name`	`local` (default)	`yarn`	`yarn`
YARN	`yarn.resourcemanager.hostname`	N/A	`localhost`	*`resourcemanager`*
	`yarn.nodemanager.aux-services`	N/A	`mapreduce_shuffle`	`mapreduce_shuffle`

You can read more about configuration in "Hadoop Configuration" on page 292.

Standalone Mode

In standalone mode, there is no further action to take, since the default properties are set for standalone mode and there are no daemons to run.

Pseudodistributed Mode

In pseudodistributed mode, the configuration files should be created with the following contents and placed in the *etc/hadoop* directory. Alternatively, you can copy the *etc/hadoop* directory to another location, and then place the **-site.xml* configuration files there. The advantage of this approach is that it separates configuration settings from the installation files. If you do this, you need to set the `HADOOP_CONF_DIR` environment variable to the alternative location, or make sure you start the daemons with the `--config` option:

```
<?xml version="1.0"?>
<!-- core-site.xml -->
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost/</value>
  </property>
</configuration>
```

```
<?xml version="1.0"?>
<!-- hdfs-site.xml -->
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

```
<?xml version="1.0"?>
<!-- mapred-site.xml -->
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

```
<?xml version="1.0"?>
<!-- yarn-site.xml -->
<configuration>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>localhost</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
</configuration>
```

Configuring SSH

In pseudodistributed mode, we have to start daemons, and to do that using the supplied scripts we need to have SSH installed. Hadoop doesn't actually distinguish between pseudodistributed and fully distributed modes; it merely starts daemons on the set of hosts in the cluster (defined by the *slaves* file) by SSHing to each host and starting a daemon process. Pseudodistributed mode is just a special case of fully distributed mode in which the (single) host is localhost, so we need to make sure that we can SSH to localhost and log in without having to enter a password.

First, make sure that SSH is installed and a server is running. On Ubuntu, for example, this is achieved with:

```
% sudo apt-get install ssh
```

On Mac OS X, make sure Remote Login (under System Preferences→Sharing) is enabled for the current user (or all users).

Then, to enable passwordless login, generate a new SSH key with an empty passphrase:

```
% ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

You may also need to run *ssh-add* if you are running *ssh-agent*.

Test that you can connect with:

```
% ssh localhost
```

If successful, you should not have to type in a password.

Formatting the HDFS filesystem

Before HDFS can be used for the first time, the filesystem must be formatted. This is done by running the following command:

```
% hdfs namenode -format
```

Starting and stopping the daemons

To start the HDFS, YARN, and MapReduce daemons, type:

```
% start-dfs.sh
% start-yarn.sh
% mr-jobhistory-daemon.sh start historyserver
```

If you have placed configuration files outside the default *conf* directory, either export the `HADOOP_CONF_DIR` environment variable before running the scripts, or start the daemons with the `--config` option, which takes an absolute path to the configuration directory:

```
% start-dfs.sh --config path-to-config-directory
% start-yarn.sh --config path-to-config-directory
% mr-jobhistory-daemon.sh --config path-to-config-directory
  start historyserver
```

The following daemons will be started on your local machine: a namenode, a secondary namenode, a datanode (HDFS), a resource manager, a node manager (YARN), and a history server (MapReduce). You can check whether the daemons started successfully by looking at the logfiles in the *logs* directory (in the Hadoop installation directory) or by looking at the web UIs, at *http://localhost:50070/* for the namenode, *http://localhost:8088/* for the resource manager, and *http://localhost:19888/* for the history server. You can also use Java's `jps` command to see whether the processes are running.

Stopping the daemons is done as follows:

```
% mr-jobhistory-daemon.sh stop historyserver
% stop-yarn.sh
% stop-dfs.sh
```

Creating a user directory

Create a home directory for yourself by running the following:

```
% hadoop fs -mkdir -p /user/$USER
```

Fully Distributed Mode

Setting up a cluster of machines brings many additional considerations, so this mode is covered in Chapter 10.

APPENDIX B
Cloudera's Distribution Including Apache Hadoop

Cloudera's Distribution Including Apache Hadoop (hereafter *CDH*) is an integrated Apache Hadoop–based stack containing all the components needed for production, tested and packaged to work together. Cloudera makes the distribution available in a number of different formats: Linux packages, virtual machine images, tarballs, and tools for running CDH in the cloud. CDH is free, released under the Apache 2.0 license, and available at *http://www.cloudera.com/cdh*.

As of CDH 5, the following components are included, many of which are covered elsewhere in this book:

Apache Avro
: A cross-language data serialization library; includes rich data structures, a fast/compact binary format, and RPC

Apache Crunch
: A high-level Java API for writing data processing pipelines that can run on MapReduce or Spark

Apache DataFu (incubating)
: A library of useful statistical UDFs for doing large-scale analyses

Apache Flume
: Highly reliable, configurable streaming data collection

Apache Hadoop
: Highly scalable data storage (HDFS), resource management (YARN), and processing (MapReduce)

Apache HBase
: Column-oriented real-time database for random read/write access

Apache Hive
: SQL-like queries and tables for large datasets

Hue
: Web UI to make it easy to work with Hadoop data

Cloudera Impala
: Interactive, low-latency SQL queries on HDFS or HBase

Kite SDK
: APIs, examples, and docs for building apps on top of Hadoop

Apache Mahout
: Scalable machine-learning and data-mining algorithms

Apache Oozie
: Workflow scheduler for interdependent Hadoop jobs

Apache Parquet (incubating)
: An efficient columnar storage format for nested data

Apache Pig
: Data flow language for exploring large datasets

Cloudera Search
: Free-text, Google-style search of Hadoop data

Apache Sentry (incubating)
: Granular, role-based access control for Hadoop users

Apache Spark
: A cluster computing framework for large-scale in-memory data processing in Scala, Java, and Python

Apache Sqoop
: Efficient transfer of data between structured data stores (like relational databases) and Hadoop

Apache ZooKeeper
: Highly available coordination service for distributed applications

Cloudera also provides *Cloudera Manager* for deploying and operating Hadoop clusters running CDH.

To download CDH and Cloudera Manager, visit *http://www.cloudera.com/downloads*.

APPENDIX C
Preparing the NCDC Weather Data

This appendix gives a runthrough of the steps taken to prepare the raw weather datafiles so they are in a form that is amenable to analysis using Hadoop. If you want to get a copy of the data to process using Hadoop, you can do so by following the instructions given at the website that accompanies this book. The rest of this appendix explains how the raw weather datafiles were processed.

The raw data is provided as a collection of tar files, compressed with bzip2. Each year's worth of readings comes in a separate file. Here's a partial directory listing of the files:

```
1901.tar.bz2
1902.tar.bz2
1903.tar.bz2
...
2000.tar.bz2
```

Each tar file contains a file for each weather station's readings for the year, compressed with gzip. (The fact that the files in the archive are compressed makes the bzip2 compression on the archive itself redundant.) For example:

```
% tar jxf 1901.tar.bz2
% ls 1901 | head
029070-99999-1901.gz
029500-99999-1901.gz
029600-99999-1901.gz
029720-99999-1901.gz
029810-99999-1901.gz
227070-99999-1901.gz
```

Because there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files in Hadoop (see "Small files and CombineFileInputFormat" on page 226), so in this case, I concatenated the decompressed files for a whole year into a single file, named by the year. I did this using a MapReduce

program, to take advantage of its parallel processing capabilities. Let's take a closer look at the program.

The program has only a map function. No reduce function is needed because the map does all the file processing in parallel with no combine stage. The processing can be done with a Unix script, so the Streaming interface to MapReduce is appropriate in this case; see Example C-1.

Example C-1. Bash script to process raw NCDC datafiles and store them in HDFS

```
#!/usr/bin/env bash

# NLineInputFormat gives a single line: key is offset, value is S3 URI
read offset s3file

# Retrieve file from S3 to local disk
echo "reporter:status:Retrieving $s3file" >&2
$HADOOP_HOME/bin/hadoop fs -get $s3file .

# Un-bzip and un-tar the local file
target=`basename $s3file .tar.bz2`
mkdir -p $target
echo "reporter:status:Un-tarring $s3file to $target" >&2
tar jxf `basename $s3file` -C $target

# Un-gzip each station file and concat into one file
echo "reporter:status:Un-gzipping $target" >&2
for file in $target/*/*
do
  gunzip -c $file >> $target.all
  echo "reporter:status:Processed $file" >&2
done

# Put gzipped version into HDFS
echo "reporter:status:Gzipping $target and putting in HDFS" >&2
gzip -c $target.all | $HADOOP_HOME/bin/hadoop fs -put - gz/$target.gz
```

The input is a small text file (*ncdc_files.txt*) listing all the files to be processed (the files start out on S3, so they are referenced using S3 URIs that Hadoop understands). Here is a sample:

```
s3n://hadoopbook/ncdc/raw/isd-1901.tar.bz2
s3n://hadoopbook/ncdc/raw/isd-1902.tar.bz2
...
s3n://hadoopbook/ncdc/raw/isd-2000.tar.bz2
```

Because the input format is specified to be `NLineInputFormat`, each mapper receives one line of input, which contains the file it has to process. The processing is explained in the script, but briefly, it unpacks the bzip2 file and then concatenates each station file into a single file for the whole year. Finally, the file is gzipped and copied into HDFS. Note the use of `hadoop fs -put -` to consume from standard input.

Status messages are echoed to standard error with a `reporter:status` prefix so that they get interpreted as MapReduce status updates. This tells Hadoop that the script is making progress and is not hanging.

The script to run the Streaming job is as follows:

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -D mapred.reduce.tasks=0 \
  -D mapred.map.tasks.speculative.execution=false \
  -D mapred.task.timeout=12000000 \
  -input ncdc_files.txt \
  -inputformat org.apache.hadoop.mapred.lib.NLineInputFormat \
  -output output \
  -mapper load_ncdc_map.sh \
  -file load_ncdc_map.sh
```

I set the number of reduce tasks to zero, since this is a map-only job. I also turned off speculative execution so duplicate tasks wouldn't write the same files (although the approach discussed in "Task side-effect files" on page 207 would have worked, too). The task timeout was set to a high value so that Hadoop doesn't kill tasks that are taking a long time (for example, when unarchiving files or copying to HDFS, when no progress is reported).

Finally, the files were archived on S3 by copying them from HDFS using *distcp*.

APPENDIX D
The Old and New Java MapReduce APIs

The Java MapReduce API used throughout this book is called the "new API," and it replaces the older, functionally equivalent API. Although Hadoop ships with both the old and new MapReduce APIs, they are not compatible with each other. Should you wish to use the old API, you can, since the code for all the MapReduce examples in this book is available for the old API on the book's website (in the `oldapi` package).

There are several notable differences between the two APIs:

- The new API is in the `org.apache.hadoop.mapreduce` package (and subpackages). The old API can still be found in `org.apache.hadoop.mapred`.
- The new API favors abstract classes over interfaces, since these are easier to evolve. This means that you can add a method (with a default implementation) to an abstract class without breaking old implementations of the class.[1] For example, the `Mapper` and `Reducer` interfaces in the old API are abstract classes in the new API.
- The new API makes extensive use of *context objects* that allow the user code to communicate with the MapReduce system. The new `Context`, for example, essentially unifies the role of the `JobConf`, the `OutputCollector`, and the `Reporter` from the old API.
- In both APIs, key-value record pairs are pushed to the mapper and reducer, but in addition, the new API allows both mappers and reducers to control the execution flow by overriding the `run()` method. For example, records can be processed in batches, or the execution can be terminated before all the records have been processed. In the old API, this is possible for mappers by writing a `MapRunnable`, but no equivalent exists for reducers.

1. Technically, such a change would almost certainly break implementations that already define a method with the same signature as Jim des Rivières explains in "Evolving Java-based APIs," for all practical purposes this is treated as a compatible change.

- Job control is performed through the `Job` class in the new API, rather than the old `JobClient`, which no longer exists in the new API.
- Configuration has been unified in the new API. The old API has a special `JobConf` object for job configuration, which is an extension of Hadoop's vanilla `Configuration` object (used for configuring daemons; see "The Configuration API" on page 141). In the new API, job configuration is done through a `Configuration`, possibly via some of the helper methods on `Job`.
- Output files are named slightly differently: in the old API both map and reduce outputs are named *part-nnnnn*, whereas in the new API map outputs are named *part-m-nnnnn* and reduce outputs are named *part-r-nnnnn* (where *nnnnn* is an integer designating the part number, starting from 00000).
- User-overridable methods in the new API are declared to throw `java.lang.InterruptedException`. This means that you can write your code to be responsive to interrupts so that the framework can gracefully cancel long-running operations if it needs to.[2]
- In the new API, the `reduce()` method passes values as a `java.lang.Iterable`, rather than a `java.lang.Iterator` (as the old API does). This change makes it easier to iterate over the values using Java's `for-each` loop construct:

```
for (VALUEIN value : values) { ... }
```

Programs using the new API that were compiled against Hadoop 1 need to be recompiled to run against Hadoop 2. This is because some classes in the new MapReduce API changed to interfaces between the Hadoop 1 and Hadoop 2 releases. The symptom is an error at runtime like the following:

```
java.lang.IncompatibleClassChangeError: Found interface
org.apache.hadoop.mapreduce.TaskAttemptContext, but class was expected
```

Example D-1 shows the `MaxTemperature` application (from "Java MapReduce" on page 24) rewritten to use the old API. The differences are highlighted in bold.

2. "Java theory and practice: Dealing with InterruptedException" by Brian Goetz explains this technique in detail.

When converting your `Mapper` and `Reducer` classes to the new API, don't forget to change the signatures of the `map()` and `reduce()` methods to the new form. Just changing your class to extend the new `Mapper` or `Reducer` classes will *not* produce a compilation error or warning, because these classes provide identity forms of the `map()` and `reduce()` methods (respectively). Your mapper or reducer code, however, will not be invoked, which can lead to some hard-to-diagnose errors.

Annotating your `map()` and `reduce()` methods with the `@Override` annotation will allow the Java compiler to catch these errors.

Example D-1. Application to find the maximum temperature, using the old MapReduce API

```
public class OldMaxTemperature {

  static class OldMaxTemperatureMapper extends MapReduceBase
      implements Mapper<LongWritable, Text, Text, IntWritable> {

    private static final int MISSING = 9999;

    @Override
    public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {

      String line = value.toString();
      String year = line.substring(15, 19);
      int airTemperature;
      if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
        airTemperature = Integer.parseInt(line.substring(88, 92));
      } else {
        airTemperature = Integer.parseInt(line.substring(87, 92));
      }
      String quality = line.substring(92, 93);
      if (airTemperature != MISSING && quality.matches("[01459]")) {
        output.collect(new Text(year), new IntWritable(airTemperature));
      }
    }
  }

  static class OldMaxTemperatureReducer extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterator<IntWritable> values,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
```

```
      int maxValue = Integer.MIN_VALUE;
      while (values.hasNext()) {
        maxValue = Math.max(maxValue, values.next().get());
      }
      output.collect(key, new IntWritable(maxValue));
    }
  }

  public static void main(String[] args) throws IOException {
    if (args.length != 2) {
      System.err.println("Usage: OldMaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    JobConf conf = new JobConf(OldMaxTemperature.class);
    conf.setJobName("Max temperature");

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

    conf.setMapperClass(OldMaxTemperatureMapper.class);
    conf.setReducerClass(OldMaxTemperatureReducer.class);

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);

    JobClient.runJob(conf);
  }
}
```

About the Author

Tom White is one of the foremost experts on Hadoop. He has been an Apache Hadoop committer since February 2007, and is a member of the Apache Software Foundation. Tom is a software engineer at Cloudera, where he has worked since its foundation on the core distributions from Apache and Cloudera. Previously he was an independent Hadoop consultant, working with companies to set up, use, and extend Hadoop. He has spoken at many conferences, including ApacheCon, OSCON, and Strata. Tom has a BA in mathematics from the University of Cambridge and an MA in philosophy of science from the University of Leeds, UK. He currently lives in Wales with his family.

Colophon

The animal on the cover of *Hadoop: The Definitive Guide* is an African elephant. These members of the genus *Loxodonta* are the largest land animals on Earth (slightly larger than their cousin, the Asian elephant) and can be identified by their ears, which have been said to look somewhat like the continent of Asia. Males stand 12 feet tall at the shoulder and weigh 12,000 pounds, but they can get as big as 15,000 pounds, whereas females stand 10 feet tall and weigh 8,000–11,000 pounds. Even young elephants are very large: at birth, they already weigh approximately 200 pounds and stand about 3 feet tall.

African elephants live throughout sub-Saharan Africa. Most of the continent's elephants live on savannas and in dry woodlands. In some regions, they can be found in desert areas; in others, they are found in mountains.

The species plays an important role in the forest and savanna ecosystems in which they live. Many plant species are dependent on passing through an elephant's digestive tract before they can germinate; it is estimated that at least a third of tree species in west African forests rely on elephants in this way. Elephants grazing on vegetation also affect the structure of habitats and influence bush fire patterns. For example, under natural conditions, elephants make gaps through the rainforest, enabling the sunlight to enter, which allows the growth of various plant species. This, in turn, facilitates more abundance and more diversity of smaller animals. As a result of the influence elephants have over many plants and animals, they are often referred to as a *keystone species* because they are vital to the long-term survival of the ecosystems in which they live.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from the *Dover Pictorial Archive*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.