Analyzing the Data with Hadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job.Map and Reduce.

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

The input to our map phase is the raw NCDC data.

To visualize the way the map works, consider the following sample lines of input data (some unused columns have been dropped to fit the page, indicated by ellipses):

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

These lines are presented to the map function as the key-value pairs:

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The keys are the line offsets within the file, which we ignore in our map function.The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted asintegers):

```
(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)
```

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

```
(1949, [111, 78])
(1950, [0, 22, −11])
```

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:

(1949, 111)
(1950, 22)

This is the final output: the maximum global temperature recorded in each year.

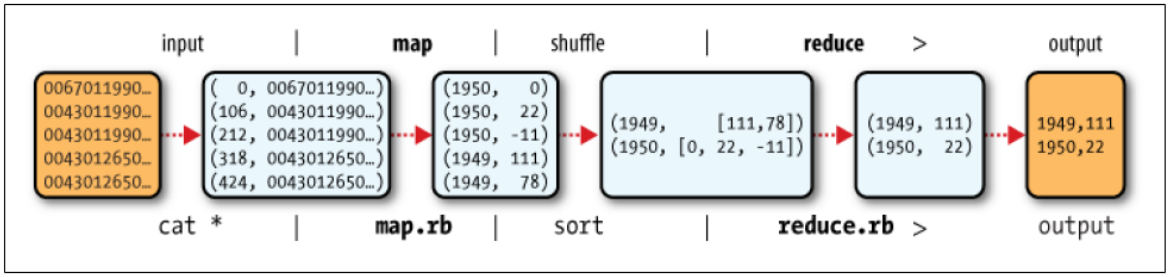


Figure 2-1. MapReduce logical data flow

Java MapReduce:*Example 2-3. Mapper for maximum temperature example*

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class MaxTemperatureMapper
extends Mapper<LongWritable, Text, Text, IntWritable>
{
	private static final int MISSING = 9999;
	@Override
	public void map(LongWritable key, Text value, Context context)
	throws IOException, InterruptedException
	{
		String line = value.toString();
		String year = line.substring(15, 19);
		int airTemperature = Integer.parseInt(line.substring(87, 92));
		context.write(new Text(year), new IntWritable(airTemperature));
	}
}
```

Example 2-4. Reducer for maximum temperature example

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class MaxTemperatureReducer
extends Reducer<Text, IntWritable, Text, IntWritable>
{
	@Override
	public void reduce(Text key, Iterable<IntWritable> values,Context context)
	throws IOException, InterruptedException
	{
```

```
                int maxValue = Integer.MIN_VALUE;
                for (IntWritable value : values)
                 {
                        maxValue = Math.max(maxValue, value.get());
                }
                context.write(key, new IntWritable(maxValue));
        }
}
```

The third piece of code runs the MapReduce job

Example 2-5. Application to find the maximum temperature in the weather dataset

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class MaxTemperature
{
        public static void main(String[] args) throws Exception {
                if (args.length != 2) {
                        System.err.println("Usage: MaxTemperature <input path> <output
                        path>");
                        System.exit(-1);
                }
                Job job = new Job();
                job.setJarByClass(MaxTemperature.class);
                job.setJobName("Max temperature");
                FileInputFormat.addInputPath(job, new Path(args[0]));
                FileOutputFormat.setOutputPath(job, new Path(args[1]));
                job.setMapperClass(MaxTemperatureMapper.class);
                job.setReducerClass(MaxTemperatureReducer.class);
                job.setOutputKeyClass(Text.class);
                job.setOutputValueClass(IntWritable.class);
                System.exit(job.waitForCompletion(true) ? 0 : 1);
        }
}
```

Analyzing the Data with Unix Tools:

Example 2-2. A program for finding the maximum recorded temperature by year from NCDC weather

records

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

for year in all/*

do
```

```
    echo -ne `basename $year .gz`"\t"
    gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
    q = substr($0, 93, 1);
    if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
    END { print max }'
done
```

The script loops through the compressed year files, first printing the year, and then processing each file using awk. The awk script extracts two fields from the data: the air temperature and the quality code. The air temperature value is turned into an integer by adding 0. Next, a test is applied to see if the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and if the quality code indicates that the reading is not suspect or erroneous. If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found. The END block is executed after all the lines in the file have been processed, and it prints the maximum value.

Here is the beginning of a run:

```
% ./max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
```

Scaling Out

For simplicity, the examples so far have used files on the local filesystem. However, to scale out, we need to store the data in a distributed filesystem, typically HDFS , to allow Hadoop to move the MapReduce computation to each machine hosting a part of the data. Let’s see how this works.

Data Flow

First, some terminology. A MapReduce *job* is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information.

Hadoop runs the job by dividing it into *tasks*, of which there are two types: *map tasks* and *reduce tasks*.

There are two types of nodes that control the job execution process: a *jobtracker* and a number of *tasktrackers*.

The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker.

Hadoop divides the input to a MapReduce job into fixed-size pieces called *input splits*, or just *splits*. Hadoop creates one map task for each split.

For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster.

So if we are processing the splits in parallel, the processing is better load-balanced if the splits are small.

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the *data locality optimization.*Sometimes, however, all three nodes hosting the HDFS block replicas for a map task's input split are running other map tasks so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer.

It should now be clear why the optimal split size is the same as the block size. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient.

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability.

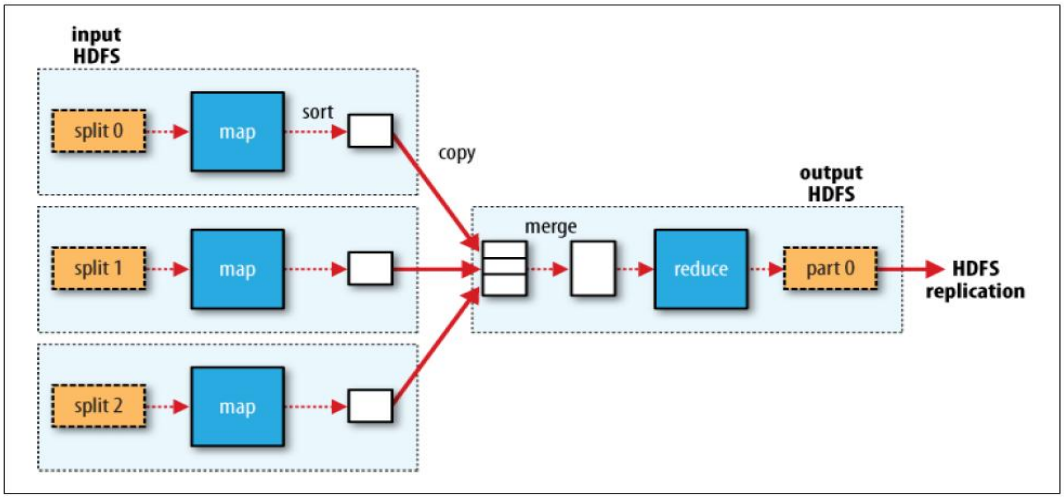


Figure 2-3. MapReduce data flow with a single reduce task

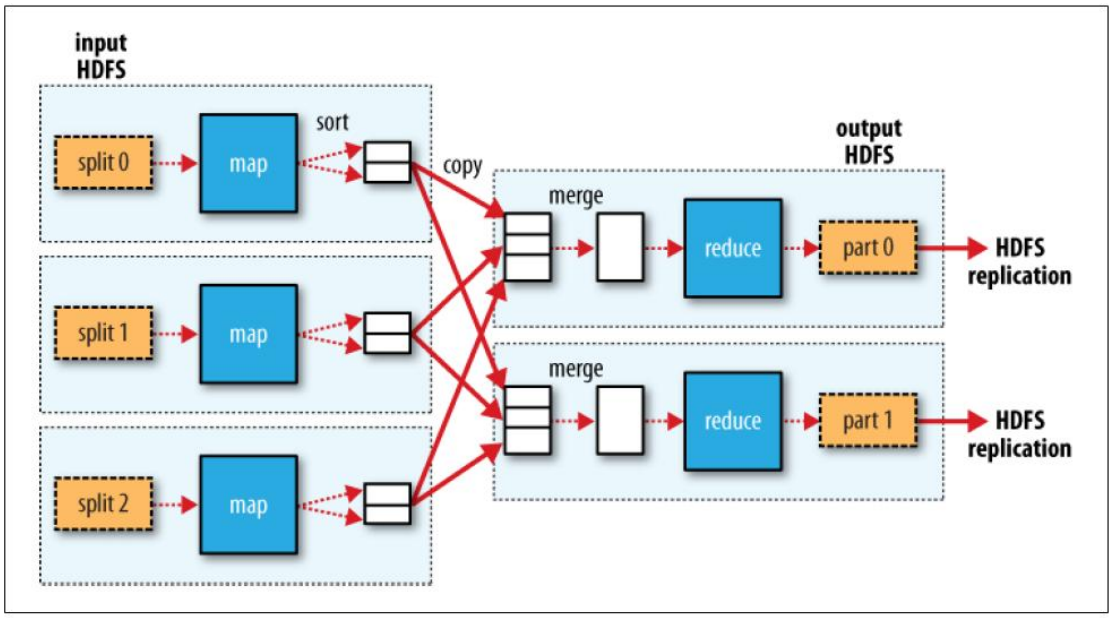


Figure 2-4. MapReduce data flow with multiple reduce tasks

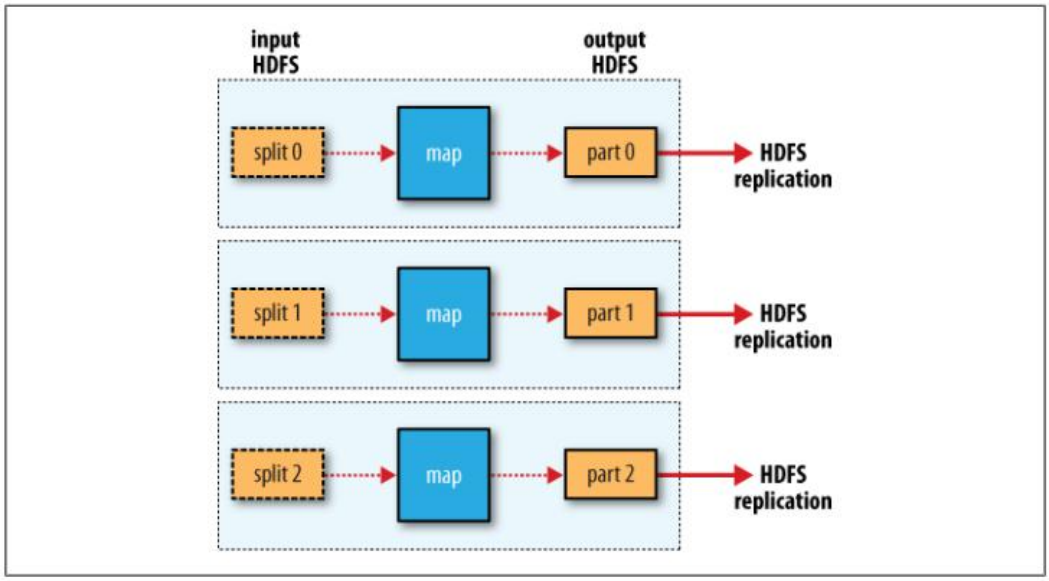


Figure 2-5. MapReduce data flow with no reduce tasks

Combiner Functions:

Combiner Functions Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function's output forms the input to the reduce function.

Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:

(1950, 0)

(1950, 20)

(1950, 10)

And the second produced:

(1950, 25)

(1950, 15)

The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])

with output:

(1950, 25)

since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce would then be called with:

(1950, [20, 25])

and the reduce would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:

max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25

The old and the new Java MapReduce APIs:

There are several notable differences between the two APIs:

- The new API favors abstract classes over interfaces. For example, the Mapper and Reducer interfaces in the old API are abstract classes in the new API.

- 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 Context,essentially unifies the role of the JobConf, the OutputCollector, and the Reporter from the old API.

- In addition, the new API allows both mappers and reducers to control the execution flow by overriding the run() method.

- Output files are named slightly differently: in the old API both map and reduce outputs are named *part-nnnnn*, while 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 zero).

- In the new API the reduce() method passes values as a java.lang.Iterable, rather than a java.lang.Iterator.

The Configuration API :

Components in Hadoop are configured using Hadoop's own configuration API. An instance of the Configuration class (found in the org.apache.hadoop.conf package) represents a collection of configuration properties and their values. Each property is named by a String, and the type of a value may be one of several types, including Java primitives such as boolean, int, long, float, and other useful types such as String, Class, java.io.File, and collections of Strings. Configurations read their properties from resources—XML files with a simple structure

for defining name-value pairs. See Example 5-1.

Example 5-1. A simple configuration file, configuration-1.xml

```
<?xml version="1.0"?>
<configuration>
        <property>
                <name>color</name>
                <value>yellow</value>
                <description>Color</description>
        </property>
        <property>
                <name>size</name>
                <value>10</value>
                <description>Size</description>
        </property>
        <property>
                <name>weight</name>
                <value>heavy</value>
                <final>true</final>
                <description>Weight</description>
        </property>


        <property>
                <name>size-weight</name>
                <value>${size},${weight}</value>
                <description>Size and weight</description>
        </property>
</configuration>
```

Assuming this configuration file is in a file called configuration-1.xml, we can access its

properties using a piece of code like this:

```
Configuration conf = new Configuration();

conf.addResource("configuration-1.xml");

assertThat(conf.get("color"), is("yellow"));

assertThat(conf.getInt("size", 0), is(10));
```

Combining Resources :

Things get interesting when more than one resource is used to define a configuration. This is used in Hadoop to separate out the default properties for the system, defined internally in a file called core-default.xml, from the site-specific overrides, in coresite.xml. The file in Example 5-2 defines the size and weight properties.

Example 5-2. A second configuration file, configuration-2.xml

```
<?xml version="1.0"?>
<configuration>
        <property>
                <name>size</name>
                <value>12</value>
         </property>
         <property>
                <name>weight</name>
                <value>light</value>
        </property>
</configuration>
```

Resources are added to a Configuration in order:

```
Configuration conf = new Configuration();

conf.addResource("configuration-1.xml");

conf.addResource("configuration-2.xml");
```

Properties defined in resources that are added later override the earlier definitions. So the size property takes its value from the second configuration file, configuration-2.xml:

```
assertThat(conf.getInt("size", 0), is(12));
```

However, properties that are marked as final cannot be overridden in later definitions.The weight property is final in the first configuration file, so the attempt to override it in the second fails, and it takes the value from the first:

```
assertThat(conf.get("weight"), is("heavy"));
```

Attempting to override final properties usually indicates a configuration error, so this results in a warning message being logged to aid diagnosis.

Variable Expansion :

Configuration properties can be defined in terms of other properties, or system properties.

CONFIGURING THE DEVELOPMENT ENVIRONMENT:

The first step is to download the version of Hadoop that you plan to use and unpack it on your development machine (this is described in Appendix A). Then, in your favourite IDE, create a new project and add all the JAR files from the top level of the unpacked distribution and from

the lib directory to the classpath. You will then be able to compile Java Hadoop programs and run them in local (standalone) mode within the IDE.

Managing Configuration:

we assume the existence of a directory called conf that contains three configuration files: hadoop-local.xml, hadoop-localhost.xml, and hadoop-cluster.xml (these are available in the example code for this book). The hadoop-local.xml file contains the default Hadoop configuration for the default filesystem and the jobtracker:

```
<?xml version="1.0"?>
<configuration>
<property>
<name>fs.default.name</name>
<value>file:///</value>
</property>
<property>
<name>mapred.job.tracker</name>
<value>local</value>
</property>
</configuration>
```

The settings in hadoop-localhost.xml point to a namenode and a jobtracker both running

on localhost:

```
<?xml version="1.0"?>
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://localhost/</value>
</property>
```

<property>

<name>mapred.job.tracker</name>

<value>localhost:8021</value>

</property>

</configuration>

Finally, hadoop-cluster.xml contains details of the cluster's namenode and jobtracker addresses. In practice, you would name the file after the name of the cluster, rather than "cluster" as we have here:

<?xml version="1.0"?>

<configuration>

<property>

<name>fs.default.name</name>

<value>hdfs://namenode/</value>

</property>

<property>

<name>mapred.job.tracker</name>

<value>jobtracker:8021</value>

</property>

</configuration>

GenericOptionsParser, Tool, and ToolRunner :

Hadoop comes with a few helper classes for making it easier to run jobs from the command line. **GenericOptionsParser** is a class that interprets common Hadoop command-line options and sets them on a Configuration object for your application to use as desired. You don't usually use GenericOptionsParser directly, as it's more convenient to implement the **Tool** interface and run your application with the **ToolRunner**, which uses GenericOptionsParser internally:

```
public interface Tool extends Configurable

{

        int run(String [] args) throws Exception;

}
```

Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. *Hadoop Streaming* uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output.

Ruby

Example 2-8. Map function for maximum temperature in Ruby

```
#!/usr/bin/env ruby
STDIN.each_line do |line|
        val = line
        year, temp, q = val[15,4], val[87,5], val[92,1]

        puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb
1950 +0000
1950 +0022
1950 -0011
1949 +0111
1949 +0078
```

```
#!/usr/bin/env ruby
last_key, max_val = nil, 0
STDIN.each_line do |line|
        key, val = line.split("\t")
        if last_key && last_key != key
                puts "#{last_key}\t#{max_val}"
                last_key, max_val = key, val.to_i
        else
                last_key, max_val = key, [max_val, val.to_i].max
        end
end
puts "#{last_key}\t#{max_val}" if last_key
```

```
% cat input/ncdc/sample.txt | ch02/src/main/ruby/max_temperature_map.rb | \
sort | ch02/src/main/ruby/max_temperature_reduce.rb
1949 111
1950 22
```

Python

Example 2-10. Map function for maximum temperature in Python

```
#!/usr/bin/env python
import re
import sys
for line in sys.stdin:
        val = line.strip()
        (year, temp, q) = (val[15:19], val[87:92], val[92:93])
        if (temp != "+9999" and re.match("[01459]", q)):
                print "%s\t%s" % (year, temp)
```

Example 2-11. Reduce function for maximum temperature in Python

```
#!/usr/bin/env python
import sys
(last_key, max_val) = (None, 0)
for line in sys.stdin:
        (key, val) = line.strip().split("\t")
        if last_key and last_key != key:
                print "%s\t%s" % (last_key, max_val)
                (last_key, max_val) = (key, int(val))
        else:
                (last_key, max_val) = (key, max(max_val, int(val)))
if last_key:
print "%s\t%s" % (last_key, max_val)
```

```
% cat input/ncdc/sample.txt | ch02/src/main/python/max_temperature_map.py | \
sort | ch02/src/main/python/max_temperature_reduce.py
1949 111
1950 22
```

WRITING A UNIT TEST:

The map and reduce functions in MapReduce are easy to test in isolation. There are several Java mock object frameworks that can help build mocks; here we use Mockito, which is noted for its clean syntax:

Mapper

The test for the mapper is shown in Example 5-4.

Example 5-4. Unit test for MaxTemperatureMapper

```
import static org.mockito.Mockito.*;

import java.io.IOException;
```

```
import org.apache.hadoop.io.*;

import org.junit.*;

public class MaxTemperatureMapperTest

{

@Test

public void processesValidRecord() throws IOException, InterruptedException

 {

MaxTemperatureMapper mapper = new MaxTemperatureMapper();

Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +

"99999V0203201N00261220001CN9999999N9-00111+99999999999");

        MaxTemperatureMapper.Context context =mock(MaxTemperatureMapper.Context.class);

mapper.map(null, value, context);

verify(context).write(new Text("1950"), new IntWritable(-11));

}

}
```

To create a mock Context,we call Mockito's mock() method (a static import), passing the class of the type we want to mock.Here we verify that Context's write() method was called with a Text object representing the year (1950) and an IntWritable representing the temperature (−1.1°C).

Example 5-5. First version of a Mapper that passes MaxTemperatureMapperTest

```
public class MaxTemperatureMapper

extends Mapper<LongWritable, Text, Text, IntWritable>

 {

@Override

public void map(LongWritable key, Text value, Context context)

throws IOException, InterruptedException

{
```

```
String line = value.toString();
String year = line.substring(15, 19);
int airTemperature = Integer.parseInt(line.substring(87, 92));
context.write(new Text(year), new IntWritable(airTemperature));
}
}
```

REDUCER:

The reducer has to find the maximum value for a given key. Here's a simple test for this feature:

```
@Test
public void returnsMaximumIntegerInValues() throws IOException,
InterruptedException
 {
MaxTemperatureReducer reducer = new MaxTemperatureReducer();
Text key = new Text("1950");
List<IntWritable> values = Arrays.asList(new IntWritable(10), new IntWritable(5));
MaxTemperatureReducer.Context context =
mock(MaxTemperatureReducer.Context.class);
reducer.reduce(key, values, context);
verify(context).write(key, new IntWritable(10));
}
```

Example 5-6. Reducer for maximum temperature example

```
public class MaxTemperatureReducer
extends Reducer<Text, IntWritable, Text, IntWritable>
{
@Override
public void reduce(Text key, Iterable<IntWritable> values,Context context)
```

```
throws IOException, InterruptedException
{
int maxValue = Integer.MIN_VALUE;
for (IntWritable value : values)
{
maxValue = Math.max(maxValue, value.get());
}
context.write(key, new IntWritable(maxValue));
}
}
```

Running a Distributed MapReduce : Running a Distributed MapReduce Job The same program will run, without alteration, on a full dataset. This is the point of MapReduce: it scales to the size of your data and the size of your hardware. Here's one data point: on a 10-node EC2 cluster running High-CPU Extra Large Instances, the program took six minutes to run.5 We'll go through the mechanics of running programs on a cluster in Chapter 5.

Running Locally on Test Data :

Now that we've got the mapper and reducer working on controlled inputs, the next step is to write a job driver and run it on some test data on a development machine. Running a Job in a Local Job Runner Using the Tool interface introduced earlier in the chapter, it's easy to write a driver to run our MapReduce job for finding the maximum temperature by year (see MaxTemperatureDriver in Example 5-7).

Example 5-7. Application to find the maximum temperature

```
public class MaxTemperatureDriver extends Configured implements Tool {
 @Override
 public int run(String[] args) throws Exception {
 if (args.length != 2) {
 System.err.printf("Usage: %s [generic options] <input> <output>\n",
 getClass().getSimpleName());
 ToolRunner.printGenericCommandUsage(System.err);
 return -1;
```

```
}

Job job = new Job(getConf(), "Max temperature");

job.setJarByClass(getClass());

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

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


job.setMapperClass(MaxTemperatureMapper.class);

job.setCombinerClass(MaxTemperatureReducer.class);

job.setReducerClass(MaxTemperatureReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(IntWritable.class);


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

}


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

int exitCode = ToolRunner.run(new MaxTemperatureDriver(), args);

System.exit(exitCode);

}

}
```

From the command line, we can run the driver by typing:

```
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \ input/ncdc/micro output
```

Testing the Driver

Apart from the flexible configuration options offered by making your application implement Tool, you also make it more testable because it allows you to inject an arbitrary Configuration. You can take advantage of this to write a test that uses a local job runner to run a job against known input data, which checks that the output is as expected.There are two approaches to doing this. The first is to use the local job runner and run the job against a test file on the local filesystem. The code in Example 5-10 gives an idea of how to do this.

Example 5-10. A test for MaxTemperatureDriver that uses a local, in-process job runner

```
@Test
public void test() throws Exception {
Configuration conf = new Configuration();
conf.set("fs.default.name", "file:///");
conf.set("mapred.job.tracker", "local");

Path input = new Path("input/ncdc/micro");
Path output = new Path("output");

FileSystem fs = FileSystem.getLocal(conf);
fs.delete(output, true); // delete old output

MaxTemperatureDriver driver = new MaxTemperatureDriver();
driver.setConf(conf);

int exitCode = driver.run(new String[] {
input.toString(), output.toString() });
assertThat(exitCode, is(0));

checkOutput(conf, output);
}
```

Running on a Cluster

Now that we are happy with the program running on a small test dataset, we are ready to try it on the full dataset on a Hadoop cluster.

Packaging

We don’t need to make any modifications to the program to run on a cluster rather than on a single machine, but we do need to package the program as a JAR file to send to the cluster. This is

conveniently achieved using Ant, using a task such as this (you can find the complete build file in the example code):

<jar destfile="hadoop-examples.jar" basedir="${classes.dir}"/>

Launching a Job

To launch the job, we need to run the driver, specifying the cluster that we want to run the job on with the -conf option (we could equally have used the -fs and -jt options):

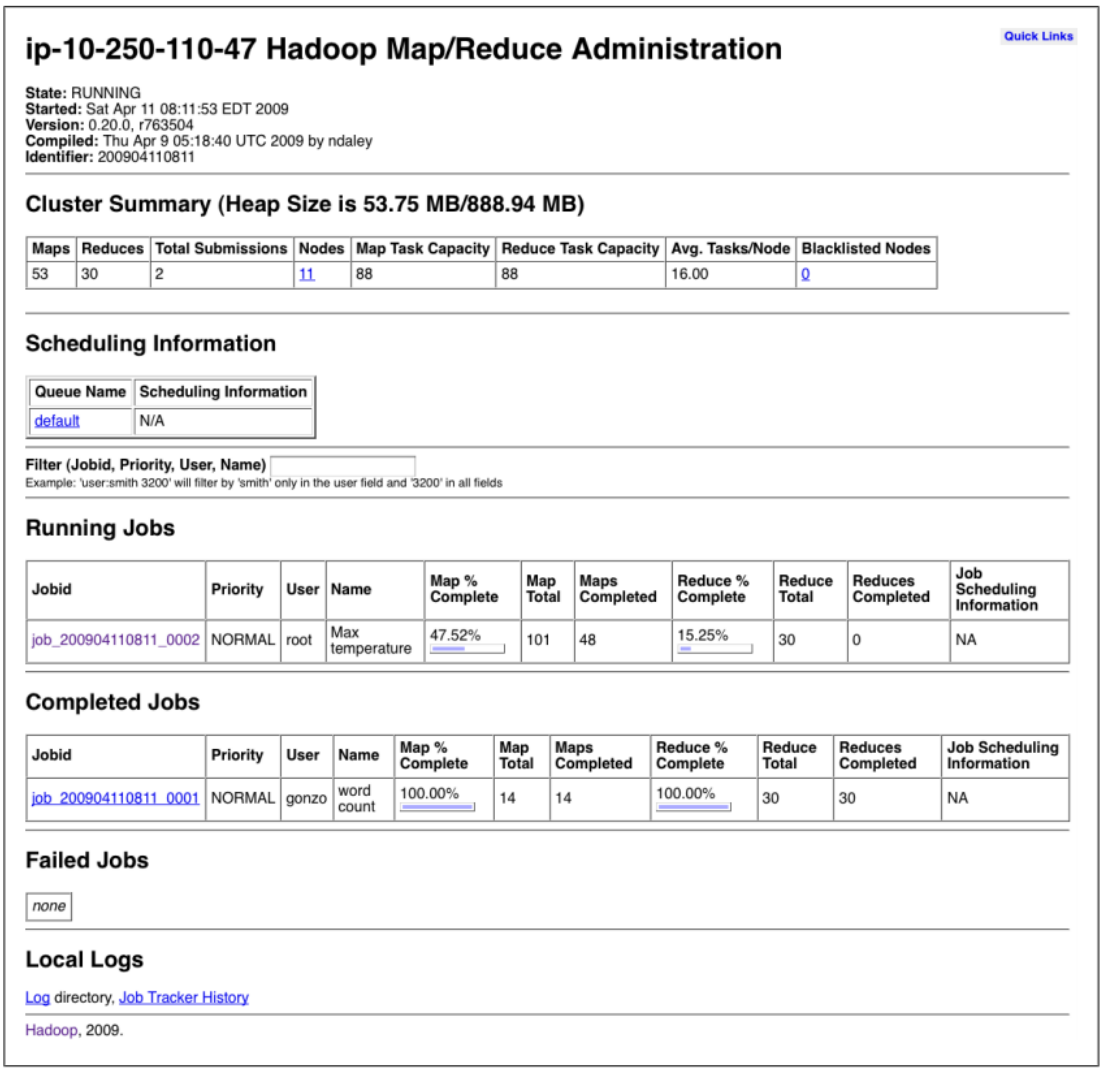
**% hadoop jar hadoop-examples.jar v3.MaxTemperatureDriver -conf conf/hadoop-cluster.xml **

input/ncdc/all max-temp

The MapReduce WebUI :

refer 164 page and 165 page.

You can find the UI at http://jobtracker-host:50030/



ip-10-250-110-47 Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2009.

Figure 5-1. Screenshot of the jobtracker page