22MCA2052 - Big Data Analytics

Module 5 – Hadoop, MapReduce

What is MapReduce?

MapReduce is a programming model for data processing. Hadoop can run MapReduce programs written in various languages. Same program can be expressed in Java, Ruby, Python, and C++. MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal.

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data, which is a good candidate for analysis with MapReduce because it is semi- structured and record-oriented.

Data Format

The data we will use is from the National Climatic Data Center. The data is stored using a line- oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, we focus on the basic elements, such as temperature, which are always present and are of fixed width.

Example 2-1 shows a sample line with some of the salient fields highlighted. The line has been split into multiple lines to show each field; in the real file, fields are packed into one line with no delimiters.

Example 2-1. Format of a National Climate Data Center record

```
0057
332130    # USAF weather station identifier
99999     # WBAN weather station identifier
19500101 # observation date
0300      # observation time
4
+51317    # latitude (degrees x 1000)
+028783   # longitude (degrees x
1000) FM-12
+0171     # elevation (meters)
99999
V020
320       # wind direction (degrees)
1         # quality
```

```
code N
0072
1
00450        # sky ceiling height (meters)
1            # quality code
C
N
010000       # visibility distance (meters)
1            # quality code
N
9
-0128        # air temperature (degrees Celsius x 10)
1            # quality code
-0139        # dew point temperature (degrees Celsius x 10)
1            # quality code
10268        # atmospheric pressure (hectopascals x 10)
1            # quality code
```

Datafiles are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

Since 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, so the data was preprocessed so that each year's readings were concatenated into a single file.

Analyzing the Data withHadoop

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job. After some local, small-scale testing, we will be able to run it on a cluster of machines.

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. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file, but as we have no need for this, we ignore it. Our map function is simple. We pull out the year and the air temperature because these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.

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, 006701199099999**1950**051507004...9999999N9+**0000**1+99999999999...)
(106, 004301199099999**1950**051512004...9999999N9+**0022**1+99999999999...)
(212, 004301199099999**1950**051518004...9999999N9-**0011**1+99999999999...)
(318, 004301265099999**1949**032412004...0500001N9+**0111**1+99999999999...)
(424, 004301265099999**1949**032418004...0500001N9+**0078**1+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 as integers):

(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.

The whole data flow is illustrated in Figure 2-1. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow.

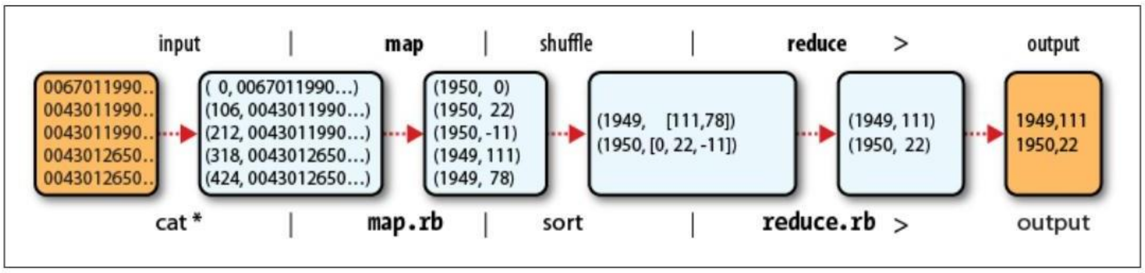


Figure 2-1. MapReduce logical data flow

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code. We need three things: a map function, a reduce function, and some code to run the job. The map function is represented by the **Mapper** class, which declares an abstract **map()** method. Example 2-3 shows the implementation of our map method.

Example 2-3. Mapper for the 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;
    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]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

The **Mapper** class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. For the present example, the input key is a long integer offset, the input value is a line of text, the outputkeyisayear, and theoutputvalueisan air temperature (an integer). Rather than use built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are foundin the **org.apache.hadoop.io** package. Here we use **LongWritable,** which corresponds to a Java **Long, Text** (like Java String), and **IntWritable** (like Java Integer).

The **map()** method is passed a key and a value. We convert the **Text** value containing the line of input into a Java String, then use its **substring()** method to extract the columns we are interested in.

The **map()** method also provides an instance of **Context** to write the output to. In this case, we write the year as a **Text** object (since we are just using it as a key), and the temperature is wrapped in an **IntWritable.** We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.

The reduce function is similarly defined using a **Reducer**, as illustrated in Example 2-4.

Example 2-4. Reducer for the 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));
  }
}
```

Again, four formal type parameters are used to specify the input and output types, this time for the **reduce** function. The input types of the reduce function must match the output types of the map function: **Text** and **IntWritable.** And in this case, the output types of the reduce function are **Text** and **IntWritable,** for a year and its maximum temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.

The third piece of code runs the MapReduce job (Example 2-5).

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);
  }
}
```

A Job object forms the specification of the job and gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster).

Having constructed a Job object, we specify the input and output paths. An input path is specified by calling the static **addInputPath()** method on **FileInputFormat**, and it can be a single file, a directory (in which case the input forms all the files in that directory), or a file pattern. As the name suggests, **addInputPath()** can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static **setOutput Path()** method on **FileOutputFormat.** It specifies a directory where the output files from the reducer Functions are written. The directory shouldn't exist before running the job because Hadoop will complain and not run the job. This precaution is to prevent data loss.

Next, we specify the map and reduce types to use via the **setMapperClass()** and **setReducerClass()** methods.

The **setOutputKeyClass()** and **setOutputValueClass()** methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, the map output types can be set using the methods **setMapOutputKeyClass()** and **setMapOutputValueClass().**

The input types are controlled via the input format, which we have not explicitly set because we are using the default **TextInputFormat.**

After setting the classes that define the map and reduce functions, we are ready to run the job. The **waitForCompletion()** method on Job submits the job and waits for it to finish. The method's Boolean argument is a verbose flag, so in this case the job writes information about its progress to the console.

The return value of the **waitForCompletion()** method is a Boolean indicating success (true) or failure (false), which we translate into the program's exit code of 0 or 1.

Scaling Out

You've seen how MapReduce works for small inputs; now it's time look at the data flow for large inputs. For simplicity, the examples so far have used files on the local filesystem. 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, the process is as follows:

Data Flow

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 systemby scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.

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, which runs the user- defined map function for each ***record*** in the split.

Having many splits means the time taken to process each split is small compared to the time to process the whole input. So if we are processing the splits in parallel, the processing is better load-balanced when the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing increases as the splits become more fine-grained.

On the other hand, if splits are too small, the overhead of managing the splits and of map task creation begins to dominate the total job execution time. 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 (for all newly created files) or specified when each file is created.

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*** because it doesn't use valuable cluster bandwidth. 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. The three possibilities are illustrated in Figure 2-2.

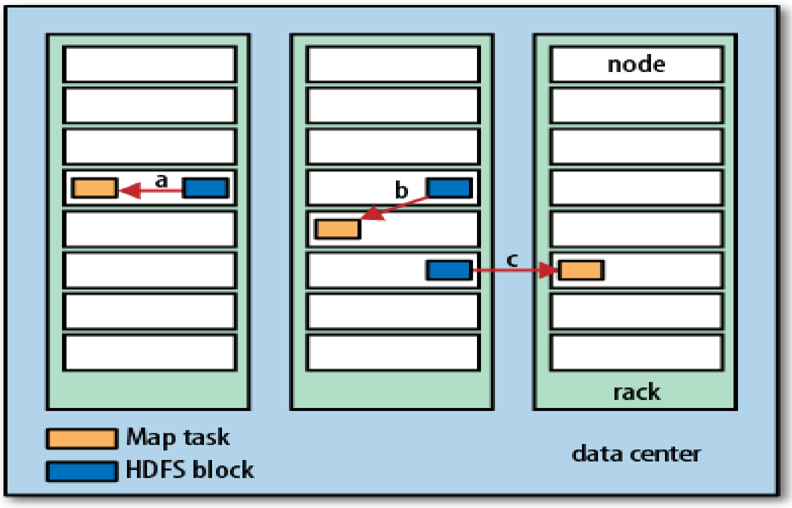


Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. 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 than running the whole map task using local data.

Map tasks write their output to the local disk, not to HDFS because Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

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. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.

The whole data flow with a single reduce task is illustrated in Figure 2-3. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes.

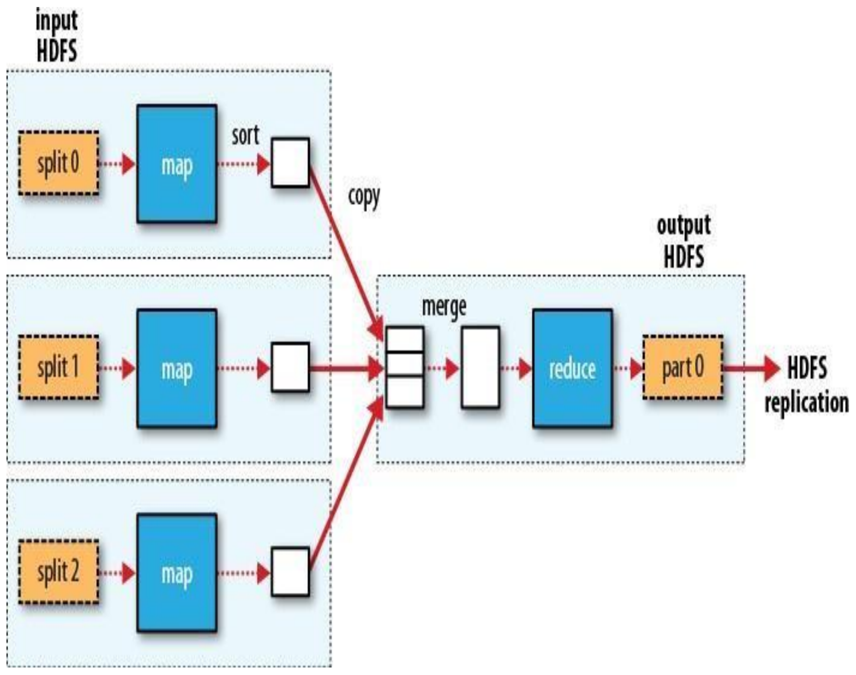


Figure 2-3. MapReduce data flow with a single reduce task

The number of reduce tasks is not governed by the size of the input, but instead is specified independently.

When there are multiple reducers, the map tasks *partition* their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time.

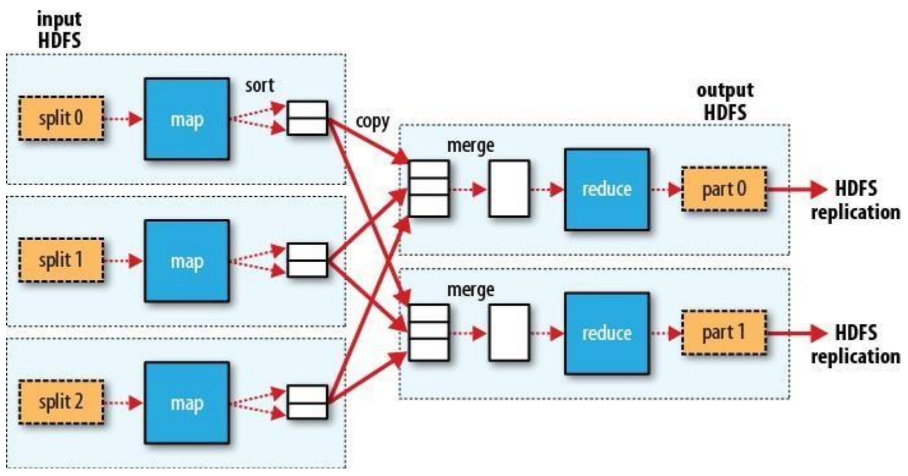


Figure 2-4. MapReduce data flow with multiple reduce tasks

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel. In this case, the only off-node data transfer is when the map tasks write to HDFS. This is shown in Fig. 2-5.

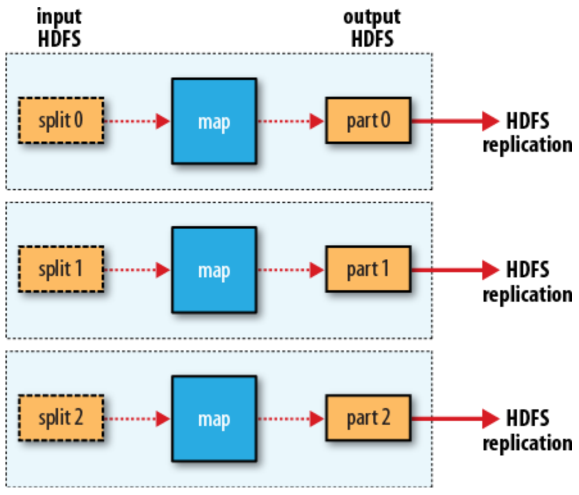


Figure 2-5. MapReduce data flow with no reduce tasks

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, and the combiner function's output forms the input to the reduce function. Because 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:

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

Not all functions possess this property. For example, if we were calculating mean temperatures, we couldn't use the mean as our combiner function, because:

$mean(0, 20, 10, 25, 15) =14$

but:

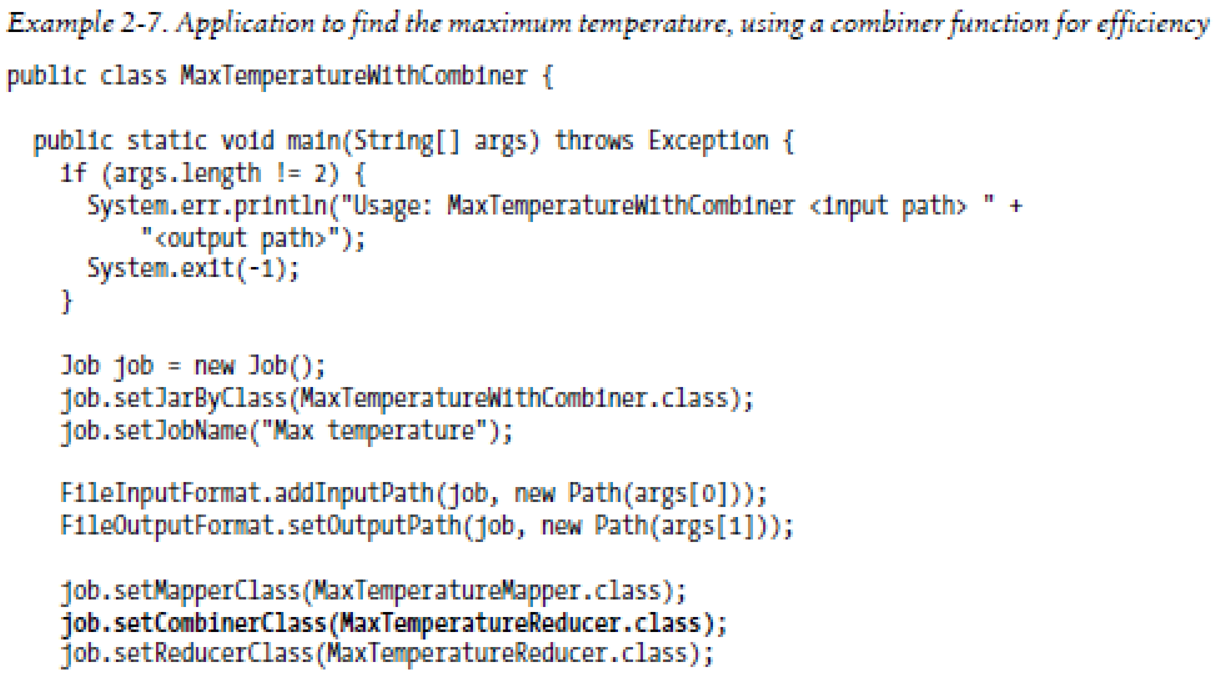
$mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15$

The combiner function doesn't replace the reduce function. But it can help cut down the amount of data shuffled between the mappers and the reducers, and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

Specifying a combiner function

Going back to the Java MapReduce program, the combiner function is defined using the Reducer class, and for this application, it is the same implementation as the reducer function in **MaxTemperatureReducer.** The only change we need to make is to set the combiner class on the **Job**.

Example 2-7. Application to find the maximum temperature, using a combiner function for efficiency.



Example 2-7. Application to find the maximum temperature, using a combiner function for efficiency

```
public class MaxTemperatureWithCombiner {

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

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    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);
```

Managing Configuration

When developing Hadoop applications, it is common to switch between running the application locally and running it on a cluster. In fact, you may have several clusters you work with, or you may have a local “pseudo distributed” cluster that you like to test on (a pseudo distributed cluster is one whose daemons all run on the local machine).

One way to accommodate these variations is to have Hadoop configuration files containing the connection settings for each cluster you run against and specify which one you are using when you run Hadoop applications or tools. As a matter of best practice, it’s recommended to keep these files outside Hadoop’s installation directory tree, as this makes it easy to switch between Hadoop versions without duplicating or losing settings.

Let us assume the existence of a directory called ***conf*** that contains three configuration files: ***hadoop-local.xml*, *hadoop-localhost.xml*, and *hadoop-cluster.xml***. Note that there is nothing special about the names of these files; they are just convenient ways to package up some configuration settings.

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
```

You can add other configuration properties to these files as needed. For example, if you wanted to set your Hadoop username for a particular cluster, you could do it in the appropriate file.

Running a MapReduce 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 a Job

The local job runner uses a single JVM to run a job, so as long as all the classes that your job needs are on its classpath, then things will just work.

In a distributed setting, things are a little more complex. For a start, a job's classes must be packagedintoa ***job JAR file*** to send to the cluster. Hadoopwill find the job JAR automatically by searching for the JAR on the driver's classpath that contains the class set in the **setJarByClass()** method(on**JobConf** or**Job**).Alternatively,ifyouwanttoset an explicit JAR file by its file path, you can use the **setJar()** method.

Creating a job JAR file is conveniently achieved using a build tool such as Ant or Maven. The following Maven command, for example, will create a JAR file called ***hadoop- examples.jar*** in the project directory containing all of the compiledclasses:

```
% mvn package -DskipTests
```

If you have a single job per JAR, you can specify the main class to run in the JAR file's manifest. If the main class is not in the manifest, it must be specified on thecommand line.

Any dependent JAR files can be packaged in a ***lib*** subdirectory in the job JAR file. Similarly, resource files can be packaged in a ***classes*** subdirectory.

The client classpath

The user's client-side classpath set by **hadoop jar *<jar>*** is made up of:

- The job JAR file
- Any JAR files in the ***lib*** directory of the job JAR file, and the ***classes*** directory (if present)
- The **classpath** defined by **HADOOP_CLASSPATH**, if set

This explains why you have to set **HADOOP_CLASSPATH** to point to dependent classes and libraries if you are running using the local job runner without a job JAR (hadoop ***CLASSNAME***).

The task classpath

On a cluster (and this includes pseudodistributed mode), map and reduce tasks run in separate JVMs, and their classpaths are *not* controlled by **HADOOP_CLASSPATH**. **HADOOP_CLASSPATH** is a client-side setting and only sets the classpath for the driver JVM, which submits the job.

Instead, the user's task classpath is comprised of the following:

- The job JAR file
- Any JAR files contained in the ***lib*** directory of the job JAR file, and the ***classes*** directory (if present)
- Any files added to the distributed cache, using the **–libjars** option, or the **addFileToClassPath()** method on **Job**.

Packaging dependencies

Given these different ways of controlling what is on the client and task classpaths, there are corresponding options for including library dependencies for a job.

- Unpack the libraries and repackage them in the job JAR.
- Package the libraries in the ***lib*** directory of the job JAR.
- Keep the libraries separate from the job JAR, and add them to the client classpath via
  HADOOP_CLASSPATH and to the task classpath via **-libjars.**

The last option, using the distributed cache, is simplest from a build point of view because dependencies don't need rebundling in the job JAR. Also, the distributed cache can mean fewer transfers of JAR files around the cluster, since files may be cached on a node between tasks.

Task classpath precedence

User JAR files are added to the end of both the client classpath and the task classpath, which in some cases can cause a dependency conflict with Hadoop's built-in libraries if Hadoop uses a different, incompatible version of a library that your code uses. Sometimes you need to be able to control the task classpath order so that your classes are picked up first. On the client side, you can force Hadoop to put the user classpath first in the search order by setting the **HADOOP_USER_CLASSPATH_FIRST** environment variable to true. For the task classpath, you can set **mapreduce.task.classpath.first** to true. Note that by setting these options you change the class loading for Hadoop framework dependencies (but only in your job), which could potentially cause the job submission or task to fail, so use these options with caution.

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:

```
% unset HADOOP_CLASSPATH
% hadoop jar hadoop-examples.jar v3.MaxTemperatureDriver \
                    -conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```

The output includes more useful information. Before the job starts, its ID is printed; this is needed whenever you want to refer to the job—in logfiles, for example—or when interrogating it via the **hadoop job** command. When the job is complete, its statistics (known as counters) are printed out. These are very useful for confirming that the job did what you expected. For example, for this job we can see that around 275 GB of input data was analyzed (Map input bytes), read from around 34 GB of compressed files on HDFS (HDFS_BYTES_READ). The input was broken into 101 gzipped files of reasonable size, so there was no problem with not being able to split them.

The MapReduce Web UI

Hadoop comes with a web UI for viewing information about your jobs. It is useful for following a job's progress while it is running, as well as finding job statistics and logs after the job has completed. You can find the UI at *http://jobtracker-host:50030/*.

The jobtracker page

A screenshot of the home page is shown in Figure 5-1. The first section of the page gives details of the Hadoop installation, such as the version number and when it was compiled, and the current state of the jobtracker (in this case, running) and when it was started.

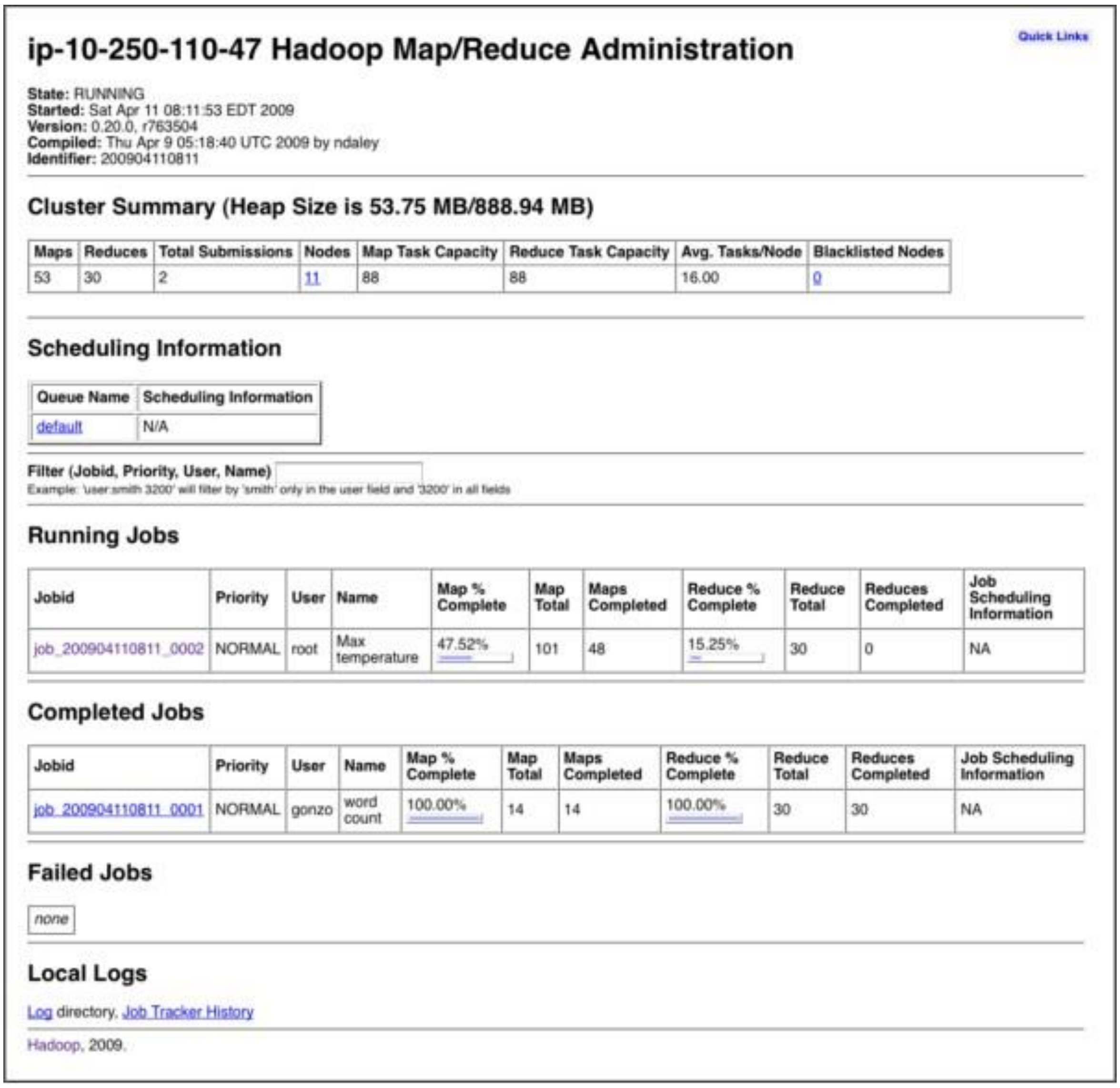
Next is a summary of the cluster, which has measures of cluster capacity and utilization. This shows the number of maps and reduces currently running on the cluster, the total number of job submissions, the number of tasktracker nodes currently available, and the cluster's capacity, in terms of the number of map and reduce slots available across the cluster ("Map Task Capacity" and "Reduce Task Capacity") and the number of available slots per node, on average. The number of tasktrackers that have been blacklisted by the jobtracker is listed as well.

Below the summary, there is a section about the job scheduler that is running (here, the default). You can also see job queues.

Further down, we see sections for running, (successfully) completed, and failed jobs. Each of these sections has a table of jobs, with a row per job that shows the job's ID, owner, name (as set in the **Job** constructor or **setJobName()** method, both of which internally set the **mapred.job.name** property), and progress information.

Finally, at the foot of the page, there are links to the jobtracker's logs and the jobtracker's history, which contains information on all the jobs that the jobtracker has run.

The main view displays only 100 jobs before consigning them to the history page. Note also that the job history is persistent, so you can find jobs here from previous runs of the jobtracker.



Quick Links

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

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

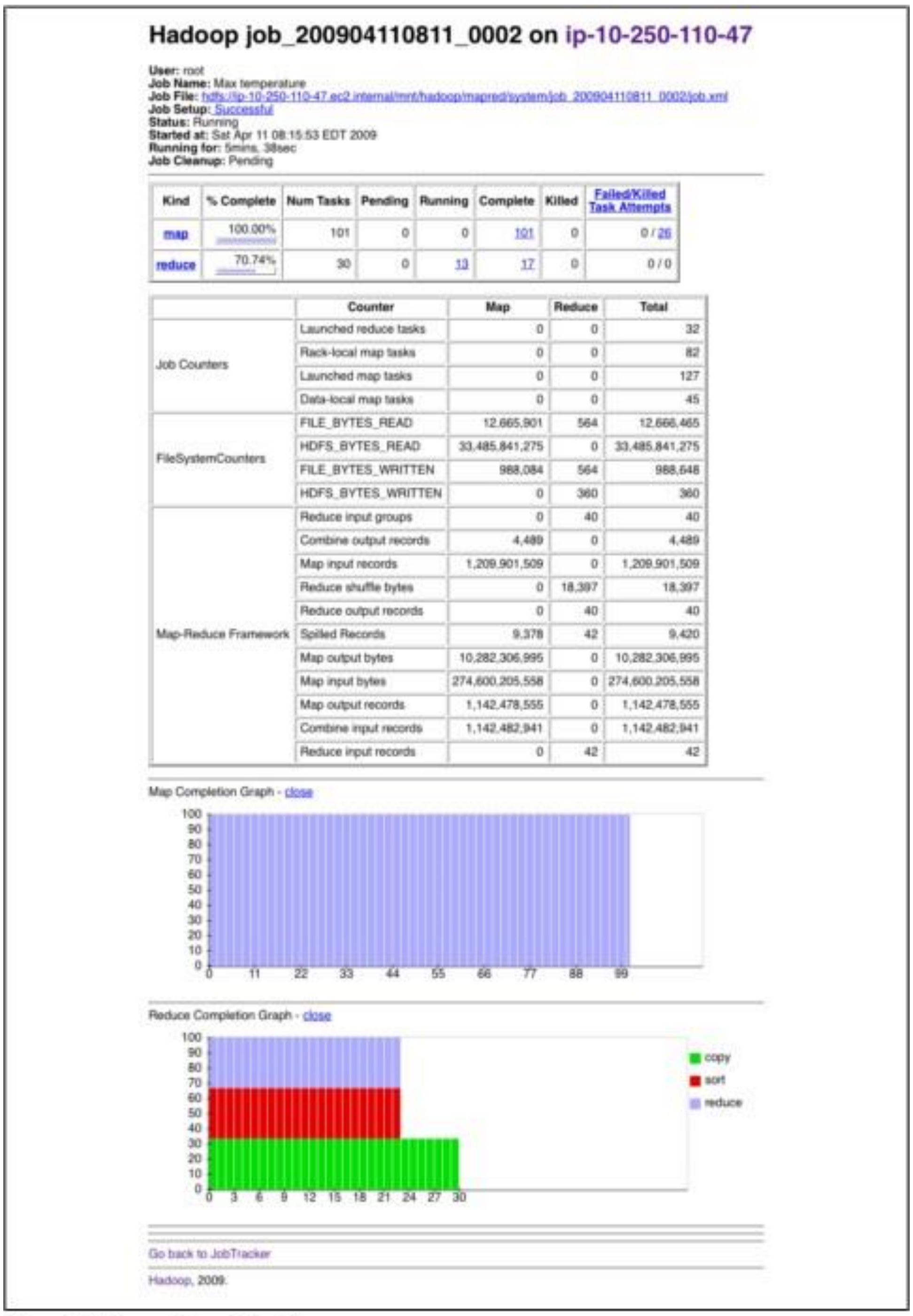
The job page

Clicking on a job ID brings you to a page for the job, illustrated in Figure 5-2. At the top of the page is a summary of the job, with basic information such as job owner and name and how long the job has been running for. The job file is the consolidated configuration file for the job, containing all the properties and their values that were in effect during the job run. If you are unsure of what a particular property was set to, you can click through to inspect the file.

While the job is running, you can monitor its progress on this page, which periodically updates itself. Below the summary is a table that shows the map progress and the reduce progress. "Num Tasks" shows the total number of map and reduce tasks for this job (a row for each). The other columns then show the state of these tasks: “Pending” (waiting to run), “Running,” “Complete” (successfully run), or “Killed” (tasks that have failed; this column would be more accurately labeled“Failed”). The final column shows the total number of failed and killed task attempts for all the map or reduce tasks for the job.

Farther down the page, you can find completion graphs for each task that show their progress graphically. The reduce completion graph is divided into the three phases of the reduce task: copy (when the map outputs are being transferred to the reduce’s tasktracker), sort (when the reduce inputs are being merged), and reduce (when the reduce function is being run to produce the final output).

In the middle of the page is a table of job counters. These are dynamically updated during the job run and provide another useful window into the job’s progress and general health.



Hadoop job_200904110811_0002 on ip-10-250-110-47

User: root
Job Name: Max temperature
Job File: hdfs://ip-10-250-110-47.ec2.internal/mnt/hadoop/mapred/system/job_200904110811_0002/job.xml
Job Setup: Successful
Status: Running
Started at: Sat Apr 11 08:15:53 EDT 2009
Running for: 5mins, 38sec
Job Cleanup: Pending

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	101	0	0	101	0	0 / 26
reduce	70.74%	30	0	13	17	0	0 / 0

	Counter	Map	Reduce	Total
Job Counters	Launched reduce tasks	0	0	32
	Rack-local map tasks	0	0	82
	Launched map tasks	0	0	127
	Data-local map tasks	0	0	45
FileSystemCounters	FILE_BYTES_READ	12,665,901	564	12,666,465
	HDFS_BYTES_READ	33,485,841,275	0	33,485,841,275
	FILE_BYTES_WRITTEN	988,084	564	988,648
	HDFS_BYTES_WRITTEN	0	360	360
Map-Reduce Framework	Reduce input groups	0	40	40
	Combine output records	4,489	0	4,489
	Map input records	1,209,901,509	0	1,209,901,509
	Reduce shuffle bytes	0	18,397	18,397
	Reduce output records	0	40	40
	Spilled Records	9,378	42	9,420
	Map output bytes	10,282,306,995	0	10,282,306,995
	Map input bytes	274,600,205,558	0	274,600,205,558
	Map output records	1,142,478,555	0	1,142,478,555
	Combine input records	1,142,482,941	0	1,142,482,941
	Reduce input records	0	42	42

Map Completion Graph - close

Reduce Completion Graph - close

Go back to JobTracker

Hadoop, 2009.

Figure 5-2. Screenshot of the job page

Retrieving the Results

Once the job is finished, there are various ways to retrieve the results. Each reducer produces one output file, so there are 30 part files named ***part-r-00000* to *part- r-00029*** in the ***max-temp*** directory.

This job produces a very small amount of output, so it is convenient to copy it from HDFS to our development machine. The **-getmerge** option to the **hadoop fs** command is useful here, as it gets all the files in the directory specified in the source pattern and merges them into a single file on the local filesystem:

```
% hadoop fs -getmerge max-temp max-temp-local
% sort max-temp-local | tail
          1991        607
          1992        605
          1993        567
          1994        568
          1995        567
          1996        561
          1997        565
          1998        568
          1999        568
          2000        558
```

We sorted the output, as the reduce output partitions are unordered. Doing a bit of postprocessing of data from MapReduce is very common, as is feeding it into analysis tools such as R, a spreadsheet, or even a relational database.

Another way of retrieving the output if it is small is to use the **-cat** option to print the output files to the console:

```
% hadoop fs -cat max-temp/*
```

On closer inspection, we see that some of the results don't look reasonable. For instance, the maximum temperature for 1951 (not shown here) is 590°C! How do we find out what's causing this? Is it corrupt input data or a bug in the program?

Debugging a Job

The way of debugging programs is via print statements, and this is certainly possible in Hadoop. However, there are complications to consider: with programs running on tens, hundreds, or thousands of nodes, how do we find and examine the output of the debug statements, which may be scattered across these nodes? For this particular case, where we are looking for an unusual case, we can use a debug statement to log to standard error, in conjunction with a message to update the task's status message to prompt us to look in the error log. The web UI makes this easy.

We also create a custom counter to count the total number of records with unreasonable temperatures in the whole dataset. This gives us valuable information about how to deal with the condition.

We add our debugging to the mapper, as opposed to the reducer, as we want to find out what the source data causing the anomalous output looks like.

If the temperature is over 100°C (represented by 1000, because temperatures are in tenths of a degree), we print a line to standard error with the suspect line, as well as update the map's status message using the **setStatus()** method on **Context** directing us to look in the log. We also increment a counter, which in Java is represented by a field o fan **enum** type. In thisprogram,wehavedefinedasinglefield**OVER_100** as a way to count the number of records with a temperature of over 100°C.

With this modification, we recompile the code, re-create the JAR file, then rerun the job, and while it's running, go to the tasks page.