Chapter 6. Developing a MapReduce Application

In **Chapter 2**, we introduced the MapReduce model. In this chapter, we look at the practical aspects of developing a MapReduce application in Hadoop.

Writing a program in MapReduce follows a certain pattern. You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect. Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working. If it fails, you can use your IDE's debugger to find the source of the problem. With this information, you can expand your unit tests to cover this case and improve your mapper or reducer as appropriate to handle such input correctly.

When the program runs as expected against the small dataset, you are ready to unleash it on a cluster. Running against the full dataset is likely to expose some more issues, which you can fix as before, by expanding your tests and altering your mapper or reducer to handle the new cases. Debugging failing programs in the cluster is a challenge, so we'll look at some common techniques to make it easier.

After the program is working, you may wish to do some tuning, first by running through some standard checks for making MapReduce programs faster and then by doing task profiling. Profiling distributed programs is not easy, but Hadoop has hooks to aid in the process.

Before we start writing a MapReduce program, however, we need to set up and configure the development environment. And to do that, we need to learn a bit about how Hadoop does configuration.

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, including Java primitives

such as `boolean`, `int`, `long`, and `float`; other useful types such as `String`, `Class`, and `java.io.File`; and collections of `String`s.

`Configuration`s read their properties from *resources*—XML files with a simple structure for defining name-value pairs. See **Example 6-1**.

Example 6-1. 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` 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));
assertThat(conf.get("breadth", "wide"), is("wide"));
```

There are a couple of things to note: type information is not stored in the XML file; instead, properties can be interpreted as a given type when they are read. Also, the `get()` methods allow you to specify a default value, which is used if the property is not defined in the XML file, as in the case of `breadth` here.

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 *core-site.xml*. The file in **Example 6-2** defines the `size` and `weight` properties.

Example 6-2. 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. Administrators mark properties as `final` in the daemon's site files that they don't want users to change in their client-side configuration files or job submission parameters.

Variable Expansion

Configuration properties can be defined in terms of other properties, or system properties. For example, the property `size-weight` in the first configuration file is defined as `${size},${weight}`, and these properties are expanded using the values found in the configuration:

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

System properties take priority over properties defined in resource files:

```
System.setProperty("size", "14");
assertThat(conf.get("size-weight"), is("14,heavy"));
```

This feature is useful for overriding properties on the command line by using `-D` *`property`*`=`*`value`* JVM arguments.

Note that although configuration properties can be defined in terms of system properties, unless system properties are redefined using configuration properties, they are not accessible through the configuration API. Hence:

```
    System.setProperty("length", "2");
    assertThat(conf.get("length"), is((String) null));
```

Setting Up the Development Environment

The first step is to create a project so you can build MapReduce programs and run them in local (standalone) mode from the command line or within your IDE. The Maven Project Object Model (POM) in **Example 6-3** shows the dependencies needed for building and testing MapReduce programs.

Example 6-3. A Maven POM for building and testing a MapReduce application

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.hadoopbook</groupId>
  <artifactId>hadoop-book-mr-dev</artifactId>
  <version>4.0</version>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <hadoop.version>2.5.1</hadoop.version>
  </properties>
  <dependencies>
    <!-- Hadoop main client artifact -->
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-client</artifactId>
      <version>${hadoop.version}</version>
    </dependency>
    <!-- Unit test artifacts -->
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.mrunit</groupId>
      <artifactId>mrunit</artifactId>
      <version>1.1.0</version>
      <classifier>hadoop2</classifier>
      <scope>test</scope>
    </dependency>
    <!-- Hadoop test artifact for running mini clusters -->
```

```
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-minicluster</artifactId>
      <version>${hadoop.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>hadoop-examples</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.5</version>
        <configuration>
          <outputDirectory>${basedir}</outputDirectory>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

The dependencies section is the interesting part of the POM. (It is straightforward to use another build tool, such as Gradle or Ant with Ivy, as long as you use the same set of dependencies defined here.) For building MapReduce jobs, you only need to have the `hadoop-client` dependency, which contains all the Hadoop client-side classes needed to interact with HDFS and MapReduce. For running unit tests, we use `junit`, and for writing MapReduce tests, we use `mrunit`. The `hadoop-minicluster` library contains the "mini-" clusters that are useful for testing with Hadoop clusters running in a single JVM.

Many IDEs can read Maven POMs directly, so you can just point them at the directory containing the *pom.xml* file and start writing code. Alternatively, you can use Maven to generate configuration files for your

IDE. For example, the following creates Eclipse configuration files so you can import the project into Eclipse:

```
% mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

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 "pseudodistributed" cluster that you like to test on (a pseudodistributed cluster is one whose daemons all run on the local machine; setting up this mode is covered in **Appendix A**).

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.

For the purposes of this book, 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). Note that there is nothing special about the names of these files; they are just convenient ways to package up some configuration settings. (Compare this to **Table A-1** in **Appendix A**, which sets out the equivalent server-side configurations.)

The *hadoop-local.xml* file contains the default Hadoop configuration for the default filesystem and the local (in-JVM) framework for running MapReduce jobs:

```
<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>file:///</value>
```

```
  </property>

  <property>
    <name>mapreduce.framework.name</name>
    <value>local</value>
  </property>

</configuration>
```

The settings in *hadoop-localhost.xml* point to a namenode and a YARN resource manager both running on localhost:

```
<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost/</value>
  </property>

  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

  <property>
    <name>yarn.resourcemanager.address</name>
    <value>localhost:8032</value>
  </property>

</configuration>
```

Finally, *hadoop-cluster.xml* contains details of the cluster's namenode and YARN resource manager 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.defaultFS</name>
    <value>hdfs://namenode/</value>
  </property>
```

```
    <property>
      <name>mapreduce.framework.name</name>
      <value>yarn</value>
    </property>

    <property>
      <name>yarn.resourcemanager.address</name>
      <value>resourcemanager:8032</value>
    </property>

  </configuration>
```

You can add other configuration properties to these files as needed.

SETTING USER IDENTITY

The user identity that Hadoop uses for permissions in HDFS is determined by running the `whoami` command on the client system. Similarly, the group names are derived from the output of running `groups`.

If, however, your Hadoop user identity is different from the name of your user account on your client machine, you can explicitly set your Hadoop username by setting the `HADOOP_USER_NAME` environment variable. You can also override user group mappings by means of the `hadoop.user.group.static.mapping.overrides` configuration property. For example, `dr.who=;preston=directors,inventors` means that the `dr.who` user is in no groups, but `preston` is in the `directors` and `inventors` groups.

You can set the user identity that the Hadoop web interfaces run as by setting the `hadoop.http.staticuser.user` property. By default, it is `dr.who`, which is not a superuser, so system files are not accessible through the web interface.

Notice that, by default, there is no authentication with this system. See **Security** for how to use Kerberos authentication with Hadoop.

With this setup, it is easy to use any configuration with the `-conf` command-line switch. For example, the following command shows a directory listing on the HDFS server running in pseudodistributed mode on localhost:

```
% hadoop fs -conf conf/hadoop-localhost.xml -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2014-09-08 10:19 input
drwxr-xr-x   - tom supergroup          0 2014-09-08 10:19 output
```

If you omit the `-conf` option, you pick up the Hadoop configuration in the *etc/hadoop* subdirectory under `$HADOOP_HOME`. Or, if `HADOOP_CONF_DIR` is set, Hadoop configuration files will be read from that location.

> **NOTE**
>
> Here's an alternative way of managing configuration settings. Copy the *etc/hadoop* directory from your Hadoop installation to another location, place the **-site.xml* configuration files there (with appropriate settings), and set the `HADOOP_CONF_DIR` environment variable to the alternative location. The main advantage of this approach is that you don't need to specify `-conf` for every command. It also allows you to isolate changes to files other than the Hadoop XML configuration files (e.g., *log4j.properties*) since the `HADOOP_CONF_DIR` directory has a copy of all the configuration files (see **Hadoop Configuration**).

Tools that come with Hadoop support the `-conf` option, but it's straightforward to make your programs (such as programs that run MapReduce jobs) support it, too, using the `Tool` interface.

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;
}
```

Example 6-4 shows a very simple implementation of `Tool` that prints the keys and values of all the properties in the `Tool`'s `Configuration` object.

Example 6-4. An example Tool implementation for printing the properties in a Configuration

```
public class ConfigurationPrinter extends Configured implements Tool {

  static {
    Configuration.addDefaultResource("hdfs-default.xml");
    Configuration.addDefaultResource("hdfs-site.xml");
    Configuration.addDefaultResource("yarn-default.xml");
    Configuration.addDefaultResource("yarn-site.xml");
    Configuration.addDefaultResource("mapred-default.xml");
    Configuration.addDefaultResource("mapred-site.xml");
  }

  @Override
  public int run(String[] args) throws Exception {
    Configuration conf = getConf();
    for (Entry<String, String> entry: conf) {
      System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
    }
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new ConfigurationPrinter(), args);
    System.exit(exitCode);
  }
}
```

We make `ConfigurationPrinter` a subclass of `Configured`, which is an implementation of the `Configurable` interface. All implementations of `Tool` need to implement `Configurable` (since `Tool` extends it), and subclassing `Configured` is often the easiest way to achieve this. The `run()` method obtains the `Configuration` using `Configurable`'s `getConf()` method and then iterates over it, printing each property to standard output.

The static block makes sure that the HDFS, YARN, and MapReduce configurations are picked up, in addition to the core ones (which `Configuration` knows about already).

`ConfigurationPrinter`'s `main()` method does not invoke its own `run()` method directly. Instead, we call `ToolRunner`'s static `run()` method, which takes care of creating a `Configuration` object for the `Tool` before calling its `run()` method. `ToolRunner` also uses a `GenericOptionsParser` to pick up any standard options specified on the command line and to set them on the `Configuration` instance. We can see the effect of picking up the properties specified in *conf/hadoop-localhost.xml* by running the following commands:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop ConfigurationPrinter -conf conf/hadoop-localhost.xml \
  | grep yarn.resourcemanager.address=
yarn.resourcemanager.address=localhost:8032
```

WHICH PROPERTIES CAN I SET?

`ConfigurationPrinter` is a useful tool for discovering what a property is set to in your environment. For a running daemon, like the namenode, you can see its configuration by viewing the */conf* page on its web server. (See **Table 10-6** to find port numbers.)

You can also see the default settings for all the public properties in Hadoop by looking in the *share/doc* directory of your Hadoop installation for files called *core-default.xml*, *hdfs-default.xml*, *yarn-default.xml*, and *mapred-default.xml*. Each property has a description that explains what it is for and what values it can be set to.

The default settings files' documentation can be found online at pages linked from **http://hadoop.apache.org/docs/current/** (look for the "Configuration" heading in the navigation). You can find the defaults for a particular Hadoop release by replacing *current* in the preceding URL with *r<version>*—for example, **http://hadoop.apache.org/docs/r2.5.0/**.

Be aware that some properties have no effect when set in the client configuration. For example, if you set `yarn.nodemanager.resource.memory-mb` in your job submission with the expectation that it would change the amount of memory available to the node managers running your job, you would be disappointed, because this property is honored only if set in the node manager's *yarn-site.xml* file. In general, you can tell the component where a property should be set by its name, so the fact that `yarn.nodemanager.resource.memory-mb` starts with `yarn.nodemanager` gives you a clue that it can be set only for the node manager daemon. This is not a hard and fast rule, however, so in some cases you may need to resort to trial and error, or even to reading the source.

Configuration property names have changed in Hadoop 2 onward, in order to give them a more regular naming structure. For example, the HDFS properties pertaining to the namenode have been changed to have a `dfs.namenode` prefix, so `dfs.name.dir` is now `dfs.namenode.name.dir`. Similarly, MapReduce properties have the `mapreduce` prefix rather than the older `mapred` prefix, so `mapred.job.name` is now `mapreduce.job.name`.

This book uses the new property names to avoid deprecation warnings. The old property names still work, however, and they are often referred to in older documentation. You can find a table listing the deprecated property names and their replacements on the **Hadoop website**.

We discuss many of Hadoop's most important configuration properties throughout this book.

`GenericOptionsParser` also allows you to set individual properties. For example:

```
% hadoop ConfigurationPrinter -D color=yellow | grep color
color=yellow
```

Here, the `-D` option is used to set the configuration property with key `color` to the value `yellow`. Options specified with `-D` take priority over properties from the configuration files. This is very useful because you can put defaults into configuration files and then override them with the `-D` option as needed. A common example of this is setting the number of reducers for a MapReduce job via `-D mapreduce.job.reduces=`*n*. This will override the number of reducers set on the cluster or set in any client-side configuration files.

The other options that `GenericOptionsParser` and `ToolRunner` support are listed in **Table 6-1**. You can find more on Hadoop's configuration API in **The Configuration API**.

WARNING

Do not confuse setting Hadoop properties using the `-D` *`property=value`* option to `GenericOptionsParser` (and `ToolRunner`) with setting JVM system properties using the `-D` *`property`*`=`*`value`* option to the `java` command. The syntax for JVM system properties does not allow any whitespace between the `D` and the property name, whereas `GenericOptionsParser` does allow whitespace.

JVM system properties are retrieved from the `java.lang.System` class, but Hadoop properties are accessible only from a `Configuration` object. So, the following command will print nothing, even though the `color` system property has been set (via `HADOOP_OPTS`), because the `System` class is not used by `ConfigurationPrinter`:

```
% HADOOP_OPTS='-Dcolor=yellow' \
  hadoop ConfigurationPrinter | grep color
```

If you want to be able to set configuration through system properties, you need to mirror the system properties of interest in the configuration file. See **Variable Expansion** for further discussion.

Table 6-1. GenericOptionsParser and ToolRunner options

Option	Description
`-D` *`property=value`*	Sets the given Hadoop configuration property to the given value. Overrides any default or site properties in the configuration and any properties set via the `-conf` option.
`-conf` *`filename ...`*	Adds the given files to the list of resources in the configuration. This is a convenient way to set site properties or to set a number of properties at once.
`-fs` *`uri`*	Sets the default filesystem to the given URI. Shortcut for `-D fs.defaultFS=`*`uri`*.
`-jt` *`host:port`*	Sets the YARN resource manager to the given host and port. (In Hadoop 1, it sets the jobtracker address, hence the option name.) Shortcut for `-D yarn.resourcemanager.address=`*`host:port`*.
`-files` *`file1,file2,...`*	Copies the specified files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and makes them available to MapReduce programs in the task's working directory. (See **Distributed Cache** for more on the distributed cache mechanism for copying files to machines in the cluster.)
`-archives` *`archive1,archive2,...`*	Copies the specified archives from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS), unarchives them, and makes them available to MapReduce programs in the task's working directory.
`-libjars` *`jar1,jar`*	Copies the specified JAR files from the local filesystem (or any filesystem if a scheme is

Option	Description
2,...	specified) to the shared filesystem used by MapReduce (usually HDFS) and adds them to the MapReduce task's classpath. This option is a useful way of shipping JAR files that a job is dependent on.

Writing a Unit Test with MRUnit

The map and reduce functions in MapReduce are easy to test in isolation, which is a consequence of their functional style. **MRUnit** is a testing library that makes it easy to pass known inputs to a mapper or a reducer and check that the outputs are as expected. MRUnit is used in conjunction with a standard test execution framework, such as JUnit, so you can run the tests for MapReduce jobs in your normal development environment. For example, all of the tests described here can be run from within an IDE by following the instructions in **Setting Up the Development Environment**.

Mapper

The test for the mapper is shown in **Example 6-5**.

Example 6-5. Unit test for MaxTemperatureMapper

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mrunit.mapreduce.MapDriver;
import org.junit.*;

public class MaxTemperatureMapperTest {

  @Test
  public void processesValidRecord() throws IOException, InterruptedException {
    Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
                                  // Year ^^^^
        "99999V0203201N00261220001CN9999999N9-00111+99999999999");
                                // Temperature ^^^^^
    new MapDriver<LongWritable, Text, Text, IntWritable>()
      .withMapper(new MaxTemperatureMapper())
      .withInput(new LongWritable(0), value)
      .withOutput(new Text("1950"), new IntWritable(-11))
      .runTest();
```

```
  }
}
```

The idea of the test is very simple: pass a weather record as input to the mapper, and check that the output is the year and temperature reading.

Since we are testing the mapper, we use MRUnit's `MapDriver`, which we configure with the mapper under test (`MaxTemperatureMapper`), the input key and value, and the expected output key (a `Text` object representing the year, 1950) and expected output value (an `IntWritable` representing the temperature, −1.1°C), before finally calling the `runTest()` method to execute the test. If the expected output values are not emitted by the mapper, MRUnit will fail the test. Notice that the input key could be set to any value because our mapper ignores it.

Proceeding in a test-driven fashion, we create a `Mapper` implementation that passes the test (see **Example 6-6**). Because we will be evolving the classes in this chapter, each is put in a different package indicating its version for ease of exposition. For example, `v1.MaxTemperatureMapper` is version 1 of `MaxTemperatureMapper`. In reality, of course, you would evolve classes without repackaging them.

Example 6-6. 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));
  }
}
```

This is a very simple implementation that pulls the year and temperature fields from the line and writes them to the `Context`. Let's add a test for missing values, which in the raw data are represented by a temperature of `+9999`:

```
@Test
public void ignoresMissingTemperatureRecord() throws IOException,
    InterruptedException {
  Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
                                  // Year ^^^^
      "99999V0203201N00261220001CN9999999N9+99991+99999999999");
                                // Temperature ^^^^^
  new MapDriver<LongWritable, Text, Text, IntWritable>()
    .withMapper(new MaxTemperatureMapper())
    .withInput(new LongWritable(0), value)
    .runTest();
}
```

A `MapDriver` can be used to check for zero, one, or more output records, according to the number of times that `withOutput()` is called. In our application, since records with missing temperatures should be filtered out, this test asserts that no output is produced for this particular input value.

The new test fails since `+9999` is not treated as a special case. Rather than putting more logic into the mapper, it makes sense to factor out a parser class to encapsulate the parsing logic; see **Example 6-7**.

Example 6-7. A class for parsing weather records in NCDC format

```
public class NcdcRecordParser {

  private static final int MISSING_TEMPERATURE = 9999;

  private String year;
  private int airTemperature;
  private String quality;

  public void parse(String record) {
    year = record.substring(15, 19);
    String airTemperatureString;
    // Remove leading plus sign as parseInt doesn't like them (pre-Java 7)
    if (record.charAt(87) == '+') {
      airTemperatureString = record.substring(88, 92);
    } else {
      airTemperatureString = record.substring(87, 92);
    }
    airTemperature = Integer.parseInt(airTemperatureString);
    quality = record.substring(92, 93);
```

```
  }

  public void parse(Text record) {
    parse(record.toString());
  }

  public boolean isValidTemperature() {
    return airTemperature != MISSING_TEMPERATURE && quality.matches("[01459]");
  }

  public String getYear() {
    return year;
  }

  public int getAirTemperature() {
    return airTemperature;
  }
}
```

The resulting mapper (version 2) is much simpler (see **Example 6-8**). It just calls the parser's `parse()` method, which parses the fields of interest from a line of input, checks whether a valid temperature was found using the `isValidTemperature()` query method, and, if it was, retrieves the year and the temperature using the getter methods on the parser. Notice that we check the quality status field as well as checking for missing temperatures in `isValidTemperature()`, to filter out poor temperature readings.

> **NOTE**
>
> Another benefit of creating a parser class is that it makes it easy to write related mappers for similar jobs without duplicating code. It also gives us the opportunity to write unit tests directly against the parser, for more targeted testing.

Example 6-8. A Mapper that uses a utility class to parse records

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private NcdcRecordParser parser = new NcdcRecordParser();

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

```
      throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      context.write(new Text(parser.getYear()),
          new IntWritable(parser.getAirTemperature()));
    }
  }
}
```

With the tests for the mapper now passing, we move on to writing the reducer.

Reducer

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

```
@Test
public void returnsMaximumIntegerInValues() throws IOException,
    InterruptedException {
  new ReduceDriver<Text, IntWritable, Text, IntWritable>()
    .withReducer(new MaxTemperatureReducer())
    .withInput(new Text("1950"),
        Arrays.asList(new IntWritable(10), new IntWritable(5)))
    .withOutput(new Text("1950"), new IntWritable(10))
    .runTest();
}
```

We construct a list of some `IntWritable` values and then verify that `MaxTemperatureReducer` picks the largest. The code in **Example 6-9** is for an implementation of `MaxTemperatureReducer` that passes the test.

Example 6-9. Reducer for the 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 Locally on Test Data

Now that we have 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 6-10**).

Example 6-10. 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);
  }
}
```

`MaxTemperatureDriver` implements the `Tool` interface, so we get the benefit of being able to set the options that `GenericOptionsParser` supports. The `run()` method constructs a `Job` object based on the tool's configuration, which it uses to launch a job. Among the possible job configuration parameters, we set the input and output file paths; the mapper, reducer, and combiner classes; and the output types (the input types are determined by the input format, which defaults to `TextInputFormat` and has `LongWritable` keys and `Text` values). It's also a good idea to set a name for the job (`Max temperature`) so that you can pick it out in the job list during execution and after it has completed. By default, the name is the name of the JAR file, which normally is not particularly descriptive.

Now we can run this application against some local files. Hadoop comes with a local job runner, a cut-down version of the MapReduce execution engine for running MapReduce jobs in a single JVM. It's designed for testing and is very convenient for use in an IDE, since you can run it in a debugger to step through the code in your mapper and reducer.

The local job runner is used if `mapreduce.framework.name` is set to `local`, which is the default.[49]

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

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \
  input/ncdc/micro output
```

Equivalently, we could use the `-fs` and `-jt` options provided by `GenericOptionsParser`:

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local input/ncdc/micro output
```

This command executes `MaxTemperatureDriver` using input from the local *input/ncdc/micro* directory, producing output in the local *output* directory. Note that although we've set `-fs` so we use the local filesystem (`file:///`), the local job runner will actually work fine against any filesystem, including HDFS (and it can be handy to do this if you have a few files that are on HDFS).

We can examine the output on the local filesystem:

```
% cat output/part-r-00000
1949    111
1950    22
```

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 6-11** gives an idea of how to do this.

Example 6-11. 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.defaultFS", "file:///");
  conf.set("mapreduce.framework.name", "local");
  conf.setInt("mapreduce.task.io.sort.mb", 1);

  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);
  }
```

The test explicitly sets `fs.defaultFS` and `mapreduce.framework.name` so it uses the local filesystem and the local job runner. It then runs the `MaxTemperatureDriver` via its `Tool` interface against a small amount of known data. At the end of the test, the `checkOutput()` method is called to compare the actual output with the expected output, line by line.

The second way of testing the driver is to run it using a "mini-" cluster. Hadoop has a set of testing classes, called `MiniDFSCluster`, `MiniMRCluster`, and `MiniYARNCluster`, that provide a programmatic way of creating in-process clusters. Unlike the local job runner, these allow testing against the full HDFS, MapReduce, and YARN machinery. Bear in mind, too, that node managers in a mini-cluster launch separate JVMs to run tasks in, which can make debugging more difficult.

TIP

You can run a mini-cluster from the command line too, with the following:

```
% hadoop jar \
  $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-*-tests.jar \
  minicluster
```

Mini-clusters are used extensively in Hadoop's own automated test suite, but they can be used for testing user code, too. Hadoop's `ClusterMapReduceTestCase` abstract class provides a useful base for writing such a test, handles the details of starting and stopping the in-process HDFS and YARN clusters in its `setUp()` and `tearDown()` methods, and generates a suitable `Configuration` object that is set up to work with them. Subclasses need only populate data in HDFS (perhaps by copying from a local file), run a MapReduce job, and confirm the output is as expected. Refer to the `MaxTemperatureDriverMiniTest` class in the example code that comes with this book for the listing.

Tests like this serve as regression tests, and are a useful repository of input edge cases and their expected results. As you encounter more test cases, you can simply add them to the input file and update the file of expected output accordingly.

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. **Chapter 10** covers how to set up a fully distributed cluster, although you can also work through this section on a pseudo-distributed 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 packaged into a *job JAR file* to send to the cluster. Hadoop will 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, if you want to set an explicit JAR file by its file path, you can use the `setJar()` method. (The JAR file path may be local or an HDFS file path.)

Creating a job JAR file is conveniently achieved using a build tool such as Ant or Maven. Given the POM in **Example 6-3**, the following Maven command will create a JAR file called *hadoop-examples.jar* in the project directory containing all of the compiled classes:

```
% 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 the command line (as we will see shortly when we run the job).

Any dependent JAR files can be packaged in a *lib* subdirectory in the job JAR file, although there are other ways to include dependencies, discussed later. Similarly, resource files can be packaged in a *classes* subdirectory. (This is analogous to a Java *Web application archive*, or WAR, file,

except in that case the JAR files go in a *WEB-INF/lib* subdirectory and classes go in a *WEB-INF/classes* subdirectory in the WAR file.)

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

Incidentally, 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 (see **Table 6-1**), or the `addFileToClassPath()` method on `DistributedCache` (old API), or `Job` (new API)

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, using the distributed cache can mean fewer transfers of JAR files around the cluster, since files may be cached on a node between tasks. (You can read more about the distributed cache .)

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.job.user.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 (we equally could have used the `-fs` and `-jt` options):

```
% unset HADOOP_CLASSPATH
% hadoop jar hadoop-examples.jar v2.MaxTemperatureDriver \
  -conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```

WARNING

We unset the `HADOOP_CLASSPATH` environment variable because we don't have any third-party dependencies for this job. If it were left set to `target/classes/` (from earlier in the chapter), Hadoop wouldn't be able to find the job JAR; it would load the `MaxTemperatureDriver` class from *target/classes* rather than the JAR, and the job would fail.

The `waitForCompletion()` method on `Job` launches the job and polls for progress, writing a line summarizing the map and reduce's progress whenever either changes. Here's the output (some lines have been removed for clarity):

```
14/09/12 06:38:11 INFO input.FileInputFormat: Total input paths to process : 101
14/09/12 06:38:11 INFO impl.YarnClientImpl: Submitted application
application_1410450250506_0003
14/09/12 06:38:12 INFO mapreduce.Job: Running job: job_1410450250506_0003
14/09/12 06:38:26 INFO mapreduce.Job:  map 0% reduce 0%
...
14/09/12 06:45:24 INFO mapreduce.Job:  map 100% reduce 100%
14/09/12 06:45:24 INFO mapreduce.Job: Job job_1410450250506_0003 completed
successfully
14/09/12 06:45:24 INFO mapreduce.Job: Counters: 49
    File System Counters
        FILE: Number of bytes read=93995
        FILE: Number of bytes written=10273563
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=33485855415
        HDFS: Number of bytes written=904
        HDFS: Number of read operations=327
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=16
    Job Counters
        Launched map tasks=101
        Launched reduce tasks=8
        Data-local map tasks=101
        Total time spent by all maps in occupied slots (ms)=5954495
        Total time spent by all reduces in occupied slots (ms)=74934
        Total time spent by all map tasks (ms)=5954495
        Total time spent by all reduce tasks (ms)=74934
        Total vcore-seconds taken by all map tasks=5954495
        Total vcore-seconds taken by all reduce tasks=74934
        Total megabyte-seconds taken by all map tasks=6097402880
        Total megabyte-seconds taken by all reduce tasks=76732416
    Map-Reduce Framework
        Map input records=1209901509
        Map output records=1143764653
        Map output bytes=10293881877
        Map output materialized bytes=14193
        Input split bytes=14140
        Combine input records=1143764772
```

```
        Combine output records=234
        Reduce input groups=100
        Reduce shuffle bytes=14193
        Reduce input records=115
        Reduce output records=100
        Spilled Records=379
        Shuffled Maps =808
        Failed Shuffles=0
        Merged Map outputs=808
        GC time elapsed (ms)=101080
        CPU time spent (ms)=5113180
        Physical memory (bytes) snapshot=60509106176
        Virtual memory (bytes) snapshot=167657209856
        Total committed heap usage (bytes)=68220878848
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=33485841275
    File Output Format Counters
        Bytes Written=90
```

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 log-files, for example—or when interrogating it via the `mapred 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 1.2 billion records were analyzed ("Map input records"), read from around 34 GB of compressed files on HDFS ("HDFS: Number of 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.

You can find out more about what the counters mean in **Built-in Counters**.

JOB, TASK, AND TASK ATTEMPT IDS

In Hadoop 2, MapReduce job IDs are generated from YARN application IDs that are created by the YARN resource manager. The format of an application ID is composed of the time that the resource manager (not the application) started and an incrementing counter maintained by the resource manager to uniquely identify the application to that instance of the resource manager. So the application with this ID:

```
application_1410450250506_0003
```

is the third (`0003` ; application IDs are 1-based) application run by the resource manager, which started at the time represented by the timestamp `1410450250506` . The counter is formatted with leading zeros to make IDs sort nicely—in directory listings, for example. However, when the counter reaches `10000` , it is not reset, resulting in longer application IDs (which don't sort so well).

The corresponding job ID is created simply by replacing the `application` prefix of an application ID with a `job` prefix:

```
job_1410450250506_0003
```

Tasks belong to a job, and their IDs are formed by replacing the `job` prefix of a job ID with a `task` prefix and adding a suffix to identify the task within the job. For example:

```
task_1410450250506_0003_m_000003
```

is the fourth (`000003` ; task IDs are 0-based) map (`m`) task of the job with ID `job_1410450250506_0003` . The task IDs are created for a job when it is initialized, so they do not necessarily dictate the order in which the tasks will be executed.

Tasks may be executed more than once, due to failure (see **Task Failure**) or speculative execution (see **Speculative Execution**), so to identify different instances of a task execution, task attempts are given unique IDs. For example:

```
attempt_1410450250506_0003_m_000003_0
```

is the first (`0` ; attempt IDs are 0-based) attempt at running task `task_1410450250506_0003_m_000003` . Task attempts are allocated during the job

run as needed, so their ordering represents the order in which they were created to run.

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://`resource-manager-host`:8088/*.

The resource manager page

A screenshot of the home page is shown in **Figure 6-1**. The "Cluster Metrics" section gives a summary of the cluster. This includes the number of applications currently running on the cluster (and in various other states), the number of resources available on the cluster ("Memory Total"), and information about node managers.

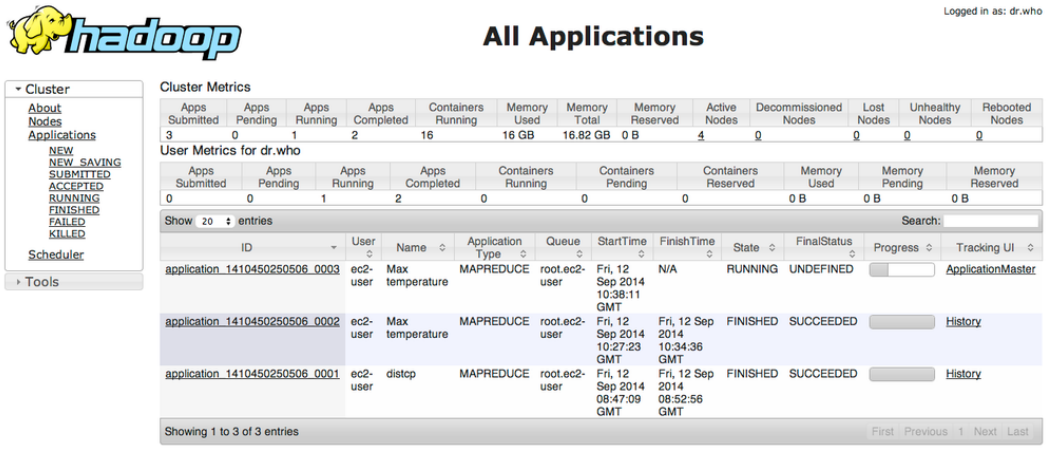

Figure 6-1. Screenshot of the resource manager page

The main table shows all the applications that have run or are currently running on the cluster. There is a search box that is useful for filtering the applications to find the ones you are interested in. The main view can show up to 100 entries per page, and the resource manager will keep up to 10,000 completed applications in memory at a time (set by `yarn.resourcemanager.max-completed-applications`), before they are only available from the job history page. Note also that the job history is persistent, so you can find jobs there from previous runs of the resource manager, too.

JOB HISTORY

Job history refers to the events and configuration for a completed MapReduce job. It is retained regardless of whether the job was successful, in an attempt to provide useful information for the user running a job.

Job history files are stored in HDFS by the MapReduce application master, in a directory set by the `mapreduce.jobhistory.done-dir` property. Job history files are kept for one week before being deleted by the system.

The history log includes job, task, and attempt events, all of which are stored in a file in JSON format. The history for a particular job may be viewed through the web UI for the job history server (which is linked to from the resource manager page) or via the command line using `mapred job -history` (which you point at the job history file).

The MapReduce job page

Clicking on the link for the "Tracking UI" takes us to the application master's web UI (or to the history page if the application has completed). In the case of MapReduce, this takes us to the job page, illustrated in Figure 6-2.

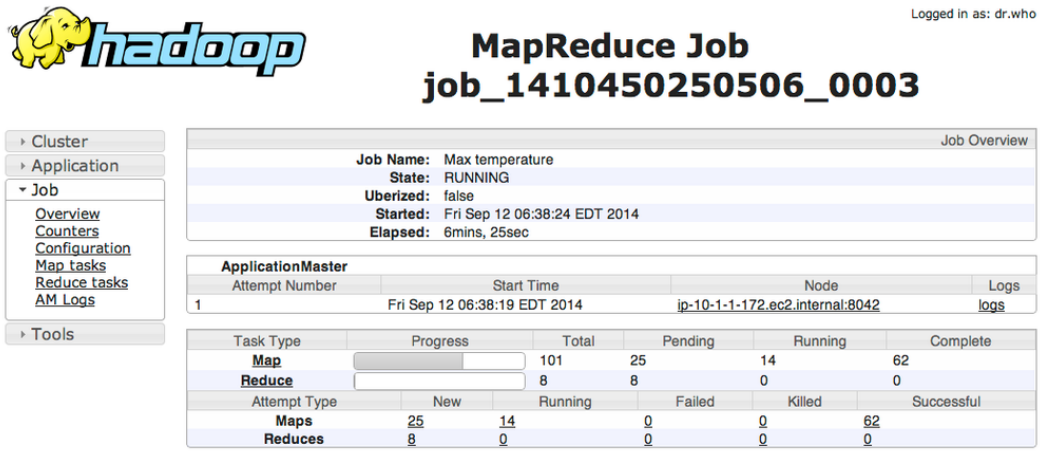

Figure 6-2. Screenshot of the job page

While the job is running, you can monitor its progress on this page. The table at the bottom shows the map progress and the reduce progress. "Total" 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," or "Complete" (successfully run).

The lower part of the table shows the total number of failed and killed task attempts for the map or reduce tasks. Task attempts may be marked

as killed if they are speculative execution duplicates, if the node they are running on dies, or if they are killed by a user. See **Task Failure** for background on task failure.

There also are a number of useful links in the navigation. For example, the "Configuration" link is to 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.

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.

NOTE

As their names suggest, a good way to think of these "part" files is as parts of the *max-temp* "file."

If the output is large (which it isn't in this case), it is important to have multiple parts so that more than one reducer can work in parallel. Usually, if a file is in this partitioned form, it can still be used easily enough—as the input to another MapReduce job, for example. In some cases, you can exploit the structure of multiple partitions to do a map-side join, for example (see **Map-Side Joins**).

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 (owing to the hash partition function). 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 plausible. 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 time-honored 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 (what we think is) an unusual case, we can use a debug statement to log to standard error, in conjunction with updating the task's status message to prompt us to look in the error log. The web UI makes this easy, as we pass:[will see].

We also create a custom counter to count the total number of records with implausible temperatures in the whole dataset. This gives us valuable information about how to deal with the condition. If it turns out to be a common occurrence, we might need to learn more about the condition and how to extract the temperature in these cases, rather than simply dropping the records. In fact, when trying to debug a job, you should always ask yourself if you can use a counter to get the information you need to find out what's happening. Even if you need to use logging or a status message, it may be useful to use a counter to gauge the extent of the problem. (There is more on counters in **Counters**.)

If the amount of log data you produce in the course of debugging is large, you have a couple of options. One is to write the information to the map's output, rather than to standard error, for analysis and aggregation by the reduce task. This approach usually necessitates structural changes to your program, so start with the other technique first. The alternative is to write a program (in MapReduce, of course) to analyze the logs produced by your job.

We add our debugging to the mapper (version 3), as opposed to the reducer, as we want to find out what the source data causing the anomalous output looks like:

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  enum Temperature {
    OVER_100
  }

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      if (airTemperature > 1000) {
        System.err.println("Temperature over 100 degrees for input: " + value);
        context.setStatus("Detected possibly corrupt record: see logs.");
        context.getCounter(Temperature.OVER_100).increment(1);
      }
      context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
    }
  }
}
```

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 updating 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 of an enum

type. In this program, we have defined a single field, `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.

The tasks and task attempts pages

The job page has a number of links for viewing the tasks in a job in more detail. For example, clicking on the "Map" link brings us to a page that lists information for all of the map tasks. The screenshot in **Figure 6-3** shows this page for the job run with our debugging statements in the "Status" column for the task.

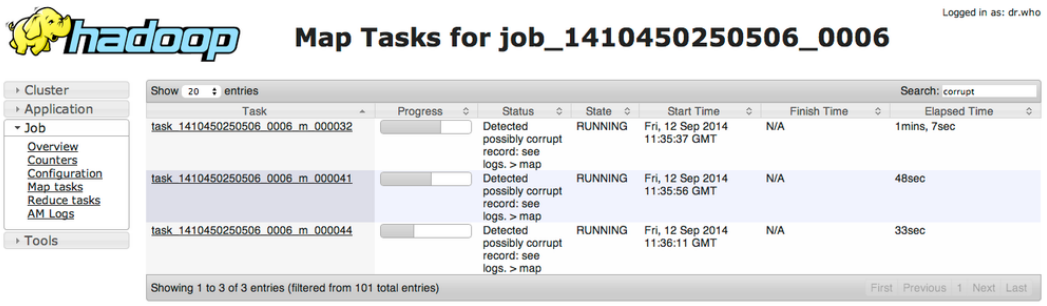

Figure 6-3. Screenshot of the tasks page

Clicking on the task link takes us to the task attempts page, which shows each task attempt for the task. Each task attempt page has links to the logfiles and counters. If we follow one of the links to the logfiles for the successful task attempt, we can find the suspect input record that we logged (the line is wrapped and truncated to fit on the page):

```
Temperature over 100 degrees for input:
033599999943318195704230200 5+37950+139117SAO  +0004RJSN V020113590031500703569999
99433201957010100005+35317+139650SAO +00899999V0200235900265007624 9N0040005...
```

This record seems to be in a different format from the others. For one thing, there are spaces in the line, which are not described in the specification.

When the job has finished, we can look at the value of the counter we defined to see how many records over 100°C there are in the whole dataset. Counters are accessible via the web UI or the command line:

```
% mapred job -counter job_1410450250506_0006 \
  'v3.MaxTemperatureMapper$Temperature' OVER_100
3
```

The `-counter` option takes the job ID, counter group name (which is the fully qualified classname here), and counter name (the enum name). There are only three malformed records in the entire dataset of over a billion records. Throwing out bad records is standard for many big data problems, although we need to be careful in this case because we are looking for an extreme value—the maximum temperature rather than an aggregate measure. Still, throwing away three records is probably not going to change the result.

Handling malformed data

Capturing input data that causes a problem is valuable, as we can use it in a test to check that the mapper does the right thing. In this MRUnit test, we check that the counter is updated for the malformed input:

```
@Test
public void parsesMalformedTemperature() throws IOException,
    InterruptedException {
  Text value = new Text("0335999999433181957042302005+37950+139117SAO  +0004" +
                                  // Year ^^^^
      "RJSN V02011359003150070356999999433201957010100005+353");
                              // Temperature ^^^^^
  Counters counters = new Counters();
  new MapDriver<LongWritable, Text, Text, IntWritable>()
    .withMapper(new MaxTemperatureMapper())
    .withInput(new LongWritable(0), value)
    .withCounters(counters)
    .runTest();
  Counter c = counters.findCounter(MaxTemperatureMapper.Temperature.MALFORMED);
  assertThat(c.getValue(), is(1L));
}
```

The record that was causing the problem is of a different format than the other lines we've seen. **Example 6-12** shows a modified program (version 4) using a parser that ignores each line with a temperature field that does not have a leading sign (plus or minus). We've also introduced a counter to measure the number of records that we are ignoring for this reason.

Example 6-12. Mapper for the maximum temperature example

```
public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  enum Temperature {
    MALFORMED
  }

  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    parser.parse(value);
    if (parser.isValidTemperature()) {
      int airTemperature = parser.getAirTemperature();
      context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
    } else if (parser.isMalformedTemperature()) {
      System.err.println("Ignoring possibly corrupt input: " + value);
      context.getCounter(Temperature.MALFORMED).increment(1);
    }
  }
}
```

Hadoop Logs

Hadoop produces logs in various places, and for various audiences. These are summarized in **Table 6-2**.

Table 6-2. Types of Hadoop logs

Logs	Primary audience	Description	Further information
System daemon logs	Administrators	Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the `HADOOP_LOG_DIR` environment variable.	**System logfiles** and **Logging**
HDFS audit logs	Administrators	A log of all HDFS requests, turned off by default. Written to the namenode's log, although this is configurable.	**Audit Logging**
MapReduce job history logs	Users	A log of the events (such as task completion) that occur in the course of running a job.	**Job History**

Logs	Primary audience	Description	Further information
		Saved centrally in HDFS.	
MapReduce task logs	Users	Each task child process produces a logfile using log4j (called *syslog*), a file for data sent to standard out (*stdout*), and a file for standard error (*stderr*). Written in the *userlogs* subdirectory of the directory defined by the `YARN_LOG_DIR` environment variable.	This section

YARN has a service for *log aggregation* that takes the task logs for completed applications and moves them to HDFS, where they are stored in a container file for archival purposes. If this service is enabled (by setting `yarn.log-aggregation-enable` to `true` on the cluster), then task logs can be viewed by clicking on the *logs* link in the task attempt web UI, or by using the `mapred job -logs` command.

By default, log aggregation is not enabled. In this case, task logs can be retrieved by visiting the node manager's web UI at *http:// `node-manager-host` :8042/logs/userlogs*.

It is straightforward to write to these logfiles. Anything written to standard output or standard error is directed to the relevant logfile. (Of course, in Streaming, standard output is used for the map or reduce output, so it will not show up in the standard output log.)

In Java, you can write to the task's *syslog* file if you wish by using the Apache Commons Logging API (or indeed any logging API that can write to log4j). This is shown in **Example 6-13**.

Example 6-13. An identity mapper that writes to standard output and also uses the Apache Commons Logging API

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.mapreduce.Mapper;

public class LoggingIdentityMapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
    extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

  private static final Log LOG = LogFactory.getLog(LoggingIdentityMapper.class);

  @Override
  @SuppressWarnings("unchecked")
  public void map(KEYIN key, VALUEIN value, Context context)
      throws IOException, InterruptedException {
    // Log to stdout file
    System.out.println("Map key: " + key);

    // Log to syslog file
    LOG.info("Map key: " + key);
    if (LOG.isDebugEnabled()) {
      LOG.debug("Map value: " + value);
    }
    context.write((KEYOUT) key, (VALUEOUT) value);
  }
}
```

The default log level is `INFO`, so `DEBUG`-level messages do not appear in the *syslog* task logfile. However, sometimes you want to see these messages. To enable this, set `mapreduce.map.log.level` or `mapreduce.reduce.log.level`, as appropriate. For example, in this case, we could set it for the mapper to see the map values in the log as follows:

```
% hadoop jar hadoop-examples.jar LoggingDriver -conf conf/hadoop-cluster.xml \
  -D mapreduce.map.log.level=DEBUG input/ncdc/sample.txt logging-out
```

There are some controls for managing the retention and size of task logs. By default, logs are deleted after a minimum of three hours (you can set this using the `yarn.nodemanager.log.retain-seconds` property, although this is ignored if log aggregation is enabled). You can also set a cap on the maximum size of each logfile using the `mapreduce.task.userlog.limit.kb` property, which is 0 by default, meaning there is no cap.

TIP

Sometimes you may need to debug a problem that you suspect is occurring in the JVM running a Hadoop command, rather than on the cluster. You can send `DEBUG`-level logs to the console by using an invocation like this:

```
% HADOOP_ROOT_LOGGER=DEBUG,console hadoop fs -text /foo/bar
```

Remote Debugging

When a task fails and there is not enough information logged to diagnose the error, you may want to resort to running a debugger for that task. This is hard to arrange when running the job on a cluster, as you don't know which node is going to process which part of the input, so you can't set up your debugger ahead of the failure. However, there are a few other options available:

Reproduce the failure locally
: Often the failing task fails consistently on a particular input. You can try to reproduce the problem locally by downloading the file that the task is failing on and running the job locally, possibly using a debugger such as Java's VisualVM.

Use JVM debugging options
: A common cause of failure is a Java out of memory error in the task JVM. You can set `mapred.child.java.opts` to include `-XX:-HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=`*`/path/to/dumps`*. This setting produces a heap dump that can be examined afterward with tools such as *jhat* or

the Eclipse Memory Analyzer. Note that the JVM options should be added to the existing memory settings specified by `mapred.child.java.opts`. These are explained in more detail in **Memory settings in YARN and MapReduce**.

Use task profiling

Java profilers give a lot of insight into the JVM, and Hadoop provides a mechanism to profile a subset of the tasks in a job. See **Profiling Tasks**.

In some cases, it's useful to keep the intermediate files for a failed task attempt for later inspection, particularly if supplementary dump or profile files are created in the task's working directory. You can set `mapreduce.task.files.preserve.failedtasks` to `true` to keep a failed task's files.

You can keep the intermediate files for successful tasks, too, which may be handy if you want to examine a task that isn't failing. In this case, set the property `mapreduce.task.files.preserve.filepattern` to a regular expression that matches the IDs of the tasks whose files you want to keep.

Another useful property for debugging is `yarn.nodemanager.delete.debug-delay-sec`, which is the number of seconds to wait to delete localized task attempt files, such as the script used to launch the task container JVM. If this is set on the cluster to a reasonably large value (e.g., `600` for 10 minutes), then you have enough time to look at the files before they are deleted.

To examine task attempt files, log into the node that the task failed on and look for the directory for that task attempt. It will be under one of the local MapReduce directories, as set by the `mapreduce.cluster.local.dir` property (covered in more detail in **Important Hadoop Daemon Properties**). If this property is a comma-separated list of directories (to spread load across the physical disks on a machine), you may need to look in all of the directories before you find the directory for that particular task attempt. The task attempt directory is in the following location:

```
mapreduce.cluster.local.dir/usercache/user/appcache/application-ID/output
  /task-attempt-ID
```

Tuning a Job

After a job is working, the question many developers ask is, “Can I make it run faster?”

There are a few Hadoop-specific “usual suspects” that are worth checking to see whether they are responsible for a performance problem. You should run through the checklist in **Table 6-3** before you start trying to profile or optimize at the task level.

Table 6-3. Tuning checklist

Area	Best practice	Further information
Number of mappers	How long are your mappers running for? If they are only running for a few seconds on average, you should see whether there's a way to have fewer mappers and make them all run longer—a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using.	**Small files and CombineFileInputFormat**
Number of reducers	Check that you are using more than a single reducer. Reduce tasks should run for five minutes or so and produce at least a block's worth of data, as a rule of thumb.	**Choosing the Number of Reducers**
Combiners	Check whether your job can take advantage of a combiner to	**Combiner Functions**

Area	Best practice	Further information
	reduce the amount of data passing through the shuffle.	
Intermediate compression	Job execution time can almost always benefit from enabling map output compression.	**Compressing map output**
Custom serialization	If you are using your own custom `Writable` objects or custom comparators, make sure you have implemented `RawComparator`.	**Implementing a RawComparator for speed**
Shuffle tweaks	The MapReduce shuffle exposes around a dozen tuning parameters for memory management, which may help you wring out the last bit of performance.	**Configuration Tuning**

Profiling Tasks

Like debugging, profiling a job running on a distributed system such as MapReduce presents some challenges. Hadoop allows you to profile a fraction of the tasks in a job and, as each task completes, pulls down the

profile information to your machine for later analysis with standard profiling tools.

Of course, it's possible, and somewhat easier, to profile a job running in the local job runner. And provided you can run with enough input data to exercise the map and reduce tasks, this can be a valuable way of improving the performance of your mappers and reducers. There are a couple of caveats, however. The local job runner is a very different environment from a cluster, and the data flow patterns are very different. Optimizing the CPU performance of your code may be pointless if your MapReduce job is I/O-bound (as many jobs are). To be sure that any tuning is effective, you should compare the new execution time with the old one running on a real cluster. Even this is easier said than done, since job execution times can vary due to resource contention with other jobs and the decisions the scheduler makes regarding task placement. To get a good idea of job execution time under these circumstances, perform a series of runs (with and without the change) and check whether any improvement is statistically significant.

It's unfortunately true that some problems (such as excessive memory use) can be reproduced only on the cluster, and in these cases the ability to profile in situ is indispensable.

The HPROF profiler

There are a number of configuration properties to control profiling, which are also exposed via convenience methods on `JobConf`. Enabling profiling is as simple as setting the property `mapreduce.task.profile` to `true`:

```
% hadoop jar hadoop-examples.jar v4.MaxTemperatureDriver \
  -conf conf/hadoop-cluster.xml \
  -D mapreduce.task.profile=true \
  input/ncdc/all max-temp
```

This runs the job as normal, but adds an `-agentlib` parameter to the Java command used to launch the task containers on the node managers. You can control the precise parameter that is added by setting the `mapreduce.task.profile.params` property. The default uses HPROF, a

profiling tool that comes with the JDK that, although basic, can give valuable information about a program's CPU and heap usage.

It doesn't usually make sense to profile all tasks in the job, so by default only those with IDs 0, 1, and 2 are profiled (for both maps and reduces). You can change this by setting `mapreduce.task.profile.maps` and `mapreduce.task.profile.reduces` to specify the range of task IDs to profile.

The profile output for each task is saved with the task logs in the *userlogs* subdirectory of the node manager's local log directory (alongside the *syslog*, *stdout*, and *stderr* files), and can be retrieved in the way described in **Hadoop Logs**, according to whether log aggregation is enabled or not.

MapReduce Workflows

So far in this chapter, you have seen the mechanics of writing a program using MapReduce. We haven't yet considered how to turn a data processing problem into the MapReduce model.

The data processing you have seen so far in this book is to solve a fairly simple problem: finding the maximum recorded temperature for given years. When the processing gets more complex, this complexity is generally manifested by having more MapReduce jobs, rather than having more complex map and reduce functions. In other words, as a rule of thumb, think about adding more jobs, rather than adding complexity to jobs.

For more complex problems, it is worth considering a higher-level language than MapReduce, such as Pig, Hive, Cascading, Crunch, or Spark. One immediate benefit is that it frees you from having to do the translation into MapReduce jobs, allowing you to concentrate on the analysis you are performing.

Finally, the book **Data-Intensive Text Processing with MapReduce** by Jimmy Lin and Chris Dyer (Morgan & Claypool Publishers, 2010) is a great resource for learning more about MapReduce algorithm design and is highly recommended.

Decomposing a Problem into MapReduce Jobs

Let's look at an example of a more complex problem that we want to translate into a MapReduce workflow.

Imagine that we want to find the mean maximum recorded temperature for every day of the year and every weather station. In concrete terms, to calculate the mean maximum daily temperature recorded by station 029070-99999, say, on January 1, we take the mean of the maximum daily temperatures for this station for January 1, 1901; January 1, 1902; and so on, up to January 1, 2000.

How can we compute this using MapReduce? The computation decomposes most naturally into two stages:

1. *Compute the maximum daily temperature for every station-date pair.*
   The MapReduce program in this case is a variant of the maximum temperature program, except that the keys in this case are a composite station-date pair, rather than just the year.
2. *Compute the mean of the maximum daily temperatures for every station-day-month key.*
   The mapper takes the output from the previous job (station-date, maximum temperature) records and projects it into (station-day-month, maximum temperature) records by dropping the year component. The reduce function then takes the mean of the maximum temperatures for each station-day-month key.

The output from the first stage looks like this for the station we are interested in (the *mean_max_daily_temp.sh* script in the examples provides an implementation in Hadoop Streaming):

```
029070-99999    19010101        0
029070-99999    19020101        -94
...
```

The first two fields form the key, and the final column is the maximum temperature from all the readings for the given station and date. The second stage averages these daily maxima over years to yield:

```
029070-99999    0101    -68
```

which is interpreted as saying the mean maximum daily temperature on January 1 for station 029070-99999 over the century is −6.8°C.

It's possible to do this computation in one MapReduce stage, but it takes more work on the part of the programmer.[50]

The arguments for having more (but simpler) MapReduce stages are that doing so leads to more composable and more maintainable mappers and reducers. Some of the case studies referred to in **Part V** cover real-world problems that were solved using MapReduce, and in each case, the data processing task is implemented using two or more MapReduce jobs. The details in that chapter are invaluable for getting a better idea of how to decompose a processing problem into a MapReduce workflow.

It's possible to make map and reduce functions even more composable than we have done. A mapper commonly performs input format parsing, projection (selecting the relevant fields), and filtering (removing records that are not of interest). In the mappers you have seen so far, we have implemented all of these functions in a single mapper. However, there is a case for splitting these into distinct mappers and chaining them into a single mapper using the `ChainMapper` library class that comes with Hadoop. Combined with a `ChainReducer`, you can run a chain of mappers, followed by a reducer and another chain of mappers, in a single MapReduce job.

JobControl

When there is more than one job in a MapReduce workflow, the question arises: how do you manage the jobs so they are executed in order? There are several approaches, and the main consideration is whether you have a linear chain of jobs or a more complex directed acyclic graph (DAG) of jobs.

For a linear chain, the simplest approach is to run each job one after another, waiting until a job completes successfully before running the next:

```
JobClient.runJob(conf1);
JobClient.runJob(conf2);
```

If a job fails, the `runJob()` method will throw an `IOException`, so later jobs in the pipeline don't get executed. Depending on your application,

you might want to catch the exception and clean up any intermediate data that was produced by any previous jobs.

The approach is similar with the new MapReduce API, except you need to examine the Boolean return value of the `waitForCompletion()` method on `Job`: `true` means the job succeeded, and `false` means it failed.

For anything more complex than a linear chain, there are libraries that can help orchestrate your workflow (although they are also suited to linear chains, or even one-off jobs). The simplest is in the `org.apache.hadoop.mapreduce.jobcontrol` package: the `JobControl` class. (There is an equivalent class in the `org.apache.hadoop.mapred.jobcontrol` package, too.) An instance of `JobControl` represents a graph of jobs to be run. You add the job configurations, then tell the `JobControl` instance the dependencies between jobs. You run the `JobControl` in a thread, and it runs the jobs in dependency order. You can poll for progress, and when the jobs have finished, you can query for all the jobs' statuses and the associated errors for any failures. If a job fails, `JobControl` won't run its dependencies.

Apache Oozie

Apache Oozie is a system for running workflows of dependent jobs. It is composed of two main parts: a *workflow engine* that stores and runs workflows composed of different types of Hadoop jobs (MapReduce, Pig, Hive, and so on), and a *coordinator engine* that runs workflow jobs based on predefined schedules and data availability. Oozie has been designed to scale, and it can manage the timely execution of thousands of workflows in a Hadoop cluster, each composed of possibly dozens of constituent jobs.

Oozie makes rerunning failed workflows more tractable, since no time is wasted running successful parts of a workflow. Anyone who has managed a complex batch system knows how difficult it can be to catch up from jobs missed due to downtime or failure, and will appreciate this feature. (Furthermore, coordinator applications representing a single data pipeline may be packaged into a *bundle* and run together as a unit.)

Unlike `JobControl`, which runs on the client machine submitting the jobs, Oozie runs as a service in the cluster, and clients submit workflow

definitions for immediate or later execution. In Oozie parlance, a workflow is a DAG of *action nodes* and *control-flow nodes*.

An action node performs a workflow task, such as moving files in HDFS; running a MapReduce, Streaming, Pig, or Hive job; performing a Sqoop import; or running an arbitrary shell script or Java program. A control-flow node governs the workflow execution between actions by allowing such constructs as conditional logic (so different execution branches may be followed depending on the result of an earlier action node) or parallel execution. When the workflow completes, Oozie can make an HTTP callback to the client to inform it of the workflow status. It is also possible to receive callbacks every time the workflow enters or exits an action node.

Defining an Oozie workflow

Workflow definitions are written in XML using the Hadoop Process Definition Language, the specification for which can be found on the **Oozie website**. **Example 6-14** shows a simple Oozie workflow definition for running a single MapReduce job.

Example 6-14. Oozie workflow definition to run the maximum temperature MapReduce job

```
<workflow-app xmlns="uri:oozie:workflow:0.1" name="max-temp-workflow">
  <start to="max-temp-mr"/>
  <action name="max-temp-mr">
    <map-reduce>
      <job-tracker>${resourceManager}</job-tracker>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="${nameNode}/user/${wf:user()}/output"/>
      </prepare>
      <configuration>
        <property>
          <name>mapred.mapper.new-api</name>
          <value>true</value>
        </property>
        <property>
          <name>mapred.reducer.new-api</name>
          <value>true</value>
        </property>
        <property>
          <name>mapreduce.job.map.class</name>
```

```
            <value>MaxTemperatureMapper</value>
          </property>
          <property>
            <name>mapreduce.job.combine.class</name>
            <value>MaxTemperatureReducer</value>
          </property>
          <property>
            <name>mapreduce.job.reduce.class</name>
            <value>MaxTemperatureReducer</value>
          </property>
          <property>
            <name>mapreduce.job.output.key.class</name>
            <value>org.apache.hadoop.io.Text</value>
          </property>
          <property>
            <name>mapreduce.job.output.value.class</name>
            <value>org.apache.hadoop.io.IntWritable</value>
          </property>
          <property>
            <name>mapreduce.input.fileinputformat.inputdir</name>
            <value>/user/${wf:user()}/input/ncdc/micro</value>
          </property>
          <property>
            <name>mapreduce.output.fileoutputformat.outputdir</name>
            <value>/user/${wf:user()}/output</value>
          </property>
        </configuration>
      </map-reduce>
      <ok to="end"/>
      <error to="fail"/>
    </action>
    <kill name="fail">
      <message>MapReduce failed, error message[${wf:errorMessage(wf:lastErrorNode()
      </message>
    </kill>
    <end name="end"/>
  </workflow-app>
```

This workflow has three control-flow nodes and one action node: a `start` control node, a `map-reduce` action node, a `kill` control node, and an `end` control node. The nodes and allowed transitions between them are shown in **Figure 6-4**.

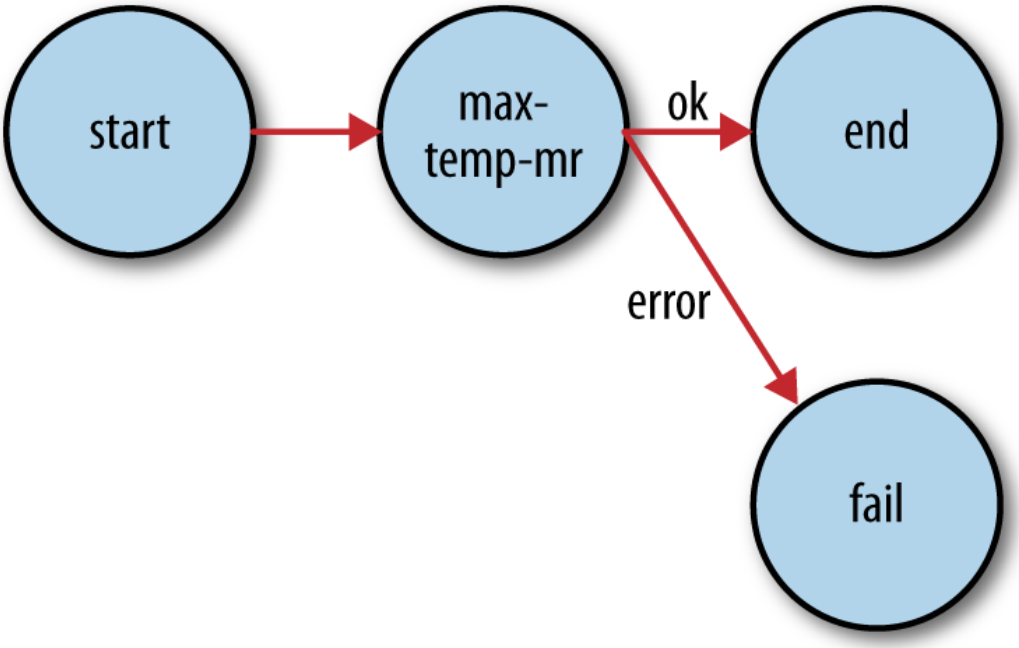

Figure 6-4. Transition diagram of an Oozie workflow

All workflows must have one `start` and one `end` node. When the workflow job starts, it transitions to the node specified by the `start` node (the `max-temp-mr` action in this example). A workflow job succeeds when it transitions to the `end` node. However, if the workflow job transitions to a `kill` node, it is considered to have failed and reports the appropriate error message specified by the `message` element in the workflow definition.

The bulk of this workflow definition file specifies the `map-reduce` action. The first two elements, `job-tracker` and `name-node`, are used to specify the YARN resource manager (or jobtracker in Hadoop 1) to submit the job to and the namenode (actually a Hadoop filesystem URI) for input and output data. Both are parameterized so that the workflow definition is not tied to a particular cluster (which makes it easy to test). The parameters are specified as workflow job properties at submission time, as we shall see later.

WARNING

Despite its name, the `job-tracker` element is used to specify a YARN resource manager address and port.

The optional `prepare` element runs before the MapReduce job and is used for directory deletion (and creation, too, if needed, although that is not shown here). By ensuring that the output directory is in a consistent state before running a job, Oozie can safely rerun the action if the job fails.

The MapReduce job to run is specified in the `configuration` element using nested elements for specifying the Hadoop configuration name-value pairs. You can view the MapReduce configuration section as a declarative replacement for the driver classes that we have used elsewhere in this book for running MapReduce programs (such as **Example 2-5**).

We have taken advantage of JSP Expression Language (EL) syntax in several places in the workflow definition. Oozie provides a set of functions for interacting with the workflow. For example, `${wf:user()}` returns the name of the user who started the current workflow job, and we use it to specify the correct filesystem path. The Oozie specification lists all the EL functions that Oozie supports.

Packaging and deploying an Oozie workflow application

A workflow application is made up of the workflow definition plus all the associated resources (such as MapReduce JAR files, Pig scripts, and so on) needed to run it. Applications must adhere to a simple directory structure, and are deployed to HDFS so that they can be accessed by Oozie. For this workflow application, we'll put all of the files in a base directory called *max-temp-workflow*, as shown diagrammatically here:

```
max-temp-workflow/
├── lib/
│   └── hadoop-examples.jar
└── workflow.xml
```

The workflow definition file *workflow.xml* must appear in the top level of this directory. JAR files containing the application's MapReduce classes are placed in the *lib* directory.

Workflow applications that conform to this layout can be built with any suitable build tool, such as Ant or Maven; you can find an example in the code that accompanies this book. Once an application has been built, it

should be copied to HDFS using regular Hadoop tools. Here is the appropriate command for this application:

```
% hadoop fs -put hadoop-examples/target/max-temp-workflow max-temp-workflow
```

Running an Oozie workflow job

Next, let's see how to run a workflow job for the application we just uploaded. For this we use the *oozie* command-line tool, a client program for communicating with an Oozie server. For convenience, we export the `OOZIE_URL` environment variable to tell the `oozie` command which Oozie server to use (here we're using one running locally):

```
% export OOZIE_URL="http://localhost:11000/oozie"
```

There are lots of subcommands for the *oozie* tool (type `oozie help` to get a list), but we're going to call the `job` subcommand with the `-run` option to run the workflow job:

```
% oozie job -config ch06-mr-dev/src/main/resources/max-temp-workflow.properties \
  -run
job: 0000001-140911033236814-oozie-oozi-W
```

The `-config` option specifies a local Java properties file containing definitions for the parameters in the workflow XML file (in this case, `nameNode` and `resourceManager`), as well as `oozie.wf.application.path`, which tells Oozie the location of the workflow application in HDFS. Here are the contents of the properties file:

```
nameNode=hdfs://localhost:8020
resourceManager=localhost:8032
oozie.wf.application.path=${nameNode}/user/${user.name}/max-temp-workflow
```

To get information about the status of the workflow job, we use the `-info` option, specifying the job ID that was printed by the run command earlier (type `oozie job` to get a list of all jobs):

```
% oozie job -info 0000001-140911033236814-oozie-oozi-W
```

The output shows the status: `RUNNING`, `KILLED`, or `SUCCEEDED`. You can also find all this information via Oozie's web UI (*http://localhost:11000/oozie*).

When the job has succeeded, we can inspect the results in the usual way:

```
% hadoop fs -cat output/part-*
1949    111
1950    22
```

This example only scratched the surface of writing Oozie workflows. The documentation on Oozie's website has information about creating more complex workflows, as well as writing and running coordinator jobs.

---

[49] In Hadoop 1, `mapred.job.tracker` determines the means of execution: `local` for the local job runner, or a colon-separated host and port pair for a jobtracker address.

---

[50] It's an interesting exercise to do this. Hint: use **Secondary Sort**.