CHAPTER 2

The Basics of a MapReduce Job

This chapter walks you through what is involved in a MapReduce job. You will be able to write and run simple stand-alone MapReduce programs by the end of the chapter.

The examples in this chapter assume the setup as described in Chapter 1. They should be explicitly run in a special local mode configuration for executing on a single machine, with no requirements for a running the Hadoop Core framework. This single machine (local) configuration is also ideal for debugging and for unit tests. The code for the examples is available from this book's details page at the Apress web site (`http://www.apress.com`). The downloadable code also includes a JAR file you can use to run the examples.

Let's start by examining the parts that make up a MapReduce job.

The Parts of a Hadoop MapReduce Job

The user configures and submits a MapReduce job (or just *job* for short) to the framework, which will decompose the job into a set of map tasks, shuffles, a sort, and a set of reduce tasks. The framework will then manage the distribution and execution of the tasks, collect the output, and report the status to the user.

The job consists of the parts shown in Figure 2-1 and listed in Table 2-1.

Table 2-1. *Parts of a MapReduce Job*

Part	Handled By
Configuration of the job	User
Input splitting and distribution	Hadoop framework
Start of the individual map tasks with their input split	Hadoop framework
Map function, called once for each input key/value pair	User
Shuffle, which partitions and sorts the per-map output	Hadoop framework
Sort, which merge sorts the shuffle output for each partition of all map outputs	Hadoop framework
Start of the individual reduce tasks, with their input partition	Hadoop framework
Reduce function, which is called once for each unique input key, with all of the input values that share that key	User
Collection of the output and storage in the configured job output directory, in *N* parts, where *N* is the number of reduce tasks	Hadoop framework

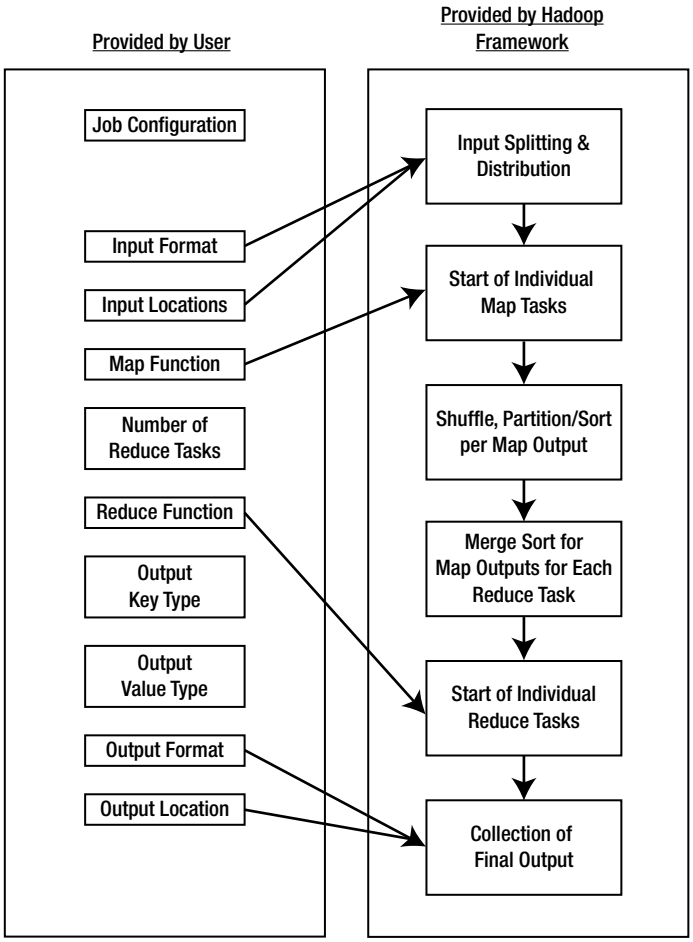

Figure 2-1. *Parts of a MapReduce job*

The user is responsible for handling the job setup, specifying the input location(s), specifying the input, and ensuring the input is in the expected format and location. The framework is responsible for distributing the job among the TaskTracker nodes of the cluster; running the map, shuffle, sort, and reduce phases; placing the output in the output directory; and informing the user of the job-completion status.

All the examples in this chapter are based on the file `MapReduceIntro.java`, shown in Listing 2-1. The job created by the code in `MapReduceIntro.java` will read all of its textual input line by line, and sort the lines based on that portion of the line before the first tab character. If there are no tab characters in the line, the sort will be based on the entire line. The `MapReduceIntro.java` file is structured to provide a simple example of configuring and running a MapReduce job.

Listing 2-1. *MapReduceIntro.java*

```
package com.apress.hadoopbook.examples.ch2;

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.RunningJob;
import org.apache.hadoop.mapred.lib.IdentityMapper;
import org.apache.hadoop.mapred.lib.IdentityReducer;
import org.apache.log4j.Logger;

/** A very simple MapReduce example that reads textual input where
 * each record is a single line, and sorts all of the input lines into
 * a single output file.
 *
 * The records are parsed into Key and Value using the first TAB
 * character as a separator. If there is no TAB character the entire
 * line is the Key. *
 *
 * @author Jason Venner
 *
 */
public class MapReduceIntro {
    protected static Logger logger = Logger.getLogger(MapReduceIntro.class);

    /**
     * Configure and run the MapReduceIntro job.
     *
     * @param args
     *            Not used.
     */
    public static void main(final String[] args) {
        try {

/** Construct the job conf object that will be used to submit this job
 * to the Hadoop framework. ensure that the jar or directory that
 * contains MapReduceIntroConfig.class is made available to all of the
 * Tasktracker nodes that will run maps or reduces for this job.
 */
            final JobConf conf = new JobConf(MapReduceIntro.class);
```

```
/**
 * Take care of some housekeeping to ensure that this simple example
 * job will run
 */
MapReduceIntroConfig.
    exampleHouseKeeping(conf,
                        MapReduceIntroConfig.getInputDirectory(),
                        MapReduceIntroConfig.getOutputDirectory());
/**
 * This section is the actual job configuration portion /**
 * Configure the inputDirectory and the type of input. In this case
 * we are stating that the input is text, and each record is a
 * single line, and the first TAB is the separator between the key
 * and the value of the record.
 */
conf.setInputFormat(KeyValueTextInputFormat.class);
FileInputFormat.setInputPaths(conf,
                        MapReduceIntroConfig.getInputDirectory());

/** Inform the framework that the mapper class will be the
 * {@link IdentityMapper}.  This class simply passes the
 * input Key Value pairs directly to its output, which in
 * our case will be the shuffle.
 */
conf.setMapperClass(IdentityMapper.class);

/** Configure the output of the job to go to the output
 * directory.  Inform the framework that the Output Key
 * and Value classes will be {@link Text} and the output
 * file format will {@link TextOutputFormat}. The
 * TextOutput format class joins produces a record of
 * output for each Key,Value pair, with the following
 * format.  Formatter.format( "%s\t%s%n", key.toString(),
 * value.toString() );.
 *
 * In addition indicate to the framework that there will be
 * 1 reduce. This results in all input keys being placed
 * into the same, single, partition, and the final output
 * being a single sorted file.
 */
FileOutputFormat.setOutputPath(conf,
                    MapReduceIntroConfig.getOutputDirectory());
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setNumReduceTasks(1);
```

```
/** Inform the framework that the reducer class will be the {@link
  * IdentityReducer}.  This class simply writes an output record key,
  * value record for each value in the key, valueset it receives as
  * input.  The value ordering is arbitrary.
  */
            conf.setReducerClass(IdentityReducer.class);

            logger .info("Launching the job.");
/** Send the job configuration to the framework and request that the
 * job be run.
 */
            final RunningJob job = JobClient.runJob(conf);
            logger.info("The job has completed.");

            if (!job.isSuccessful()) {
                logger.error("The job failed.");
                System.exit(1);
            }
            logger.info("The job completed successfully.");
            System.exit(0);
        } catch (final IOException e) {
            logger.error("The job has failed due to an IO Exception", e);
            e.printStackTrace();
        }

    }
}
```

Input Splitting

For the framework to be able to distribute pieces of the job to multiple machines, it needs to fragment the input into individual pieces, which can in turn be provided as input to the individual distributed tasks. Each fragment of input is called an *input split*. The default rules for how input splits are constructed from the actual input files are a combination of configuration parameters and the capabilities of the class that actually reads the input records. These parameters are covered in Chapter 6.

An input split will normally be a contiguous group of records from a single input file, and in this case, there will be at least *N* input splits, where *N* is the number of input files. If the number of requested map tasks is larger than this number, or the individual files are larger than the suggested fragment size, there may be multiple input splits constructed of each input file. The user has considerable control over the number of input splits. The number and size of the input splits strongly influence overall job performance.

A Simple Map Function: IdentityMapper

The Hadoop framework provides a very simple map function, called `IdentityMapper`. It is used in jobs that only need to reduce the input, and not transform the raw input. We

are going to examine the code of the `IdentityMapper` class, shown in Listing 2-2, in this section. If you have downloaded a Hadoop Core installation and followed the instructions in Chapter 1, this code is also available in the directory where you installed it, `${HADOOP_HOME}/src/mapred/org/apache/hadoop/mapred/lib/IdentityMapper.java`.

Listing 2-2. *IdentityMapper.java*

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.hadoop.mapred.lib;

import java.io.IOException;

import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;

/** Implements the identity function, mapping inputs directly to outputs. */
public class IdentityMapper<K, V>
    extends MapReduceBase implements Mapper<K, V, K, V> {

  /** The identify function.  Input key/value pair is written directly to
   * output.*/
  public void map(K key, V val,
                  OutputCollector<K, V> output, Reporter reporter)
    throws IOException {
    output.collect(key, val);
  }
}
```

The magic piece of code is the line `output.collect(key, val)`, which passes a key/value pair back to the framework for further processing.

All map functions must implement the `Mapper` interface, which guarantees that the map function will always be called with a key. The key is an instance of a `WritableComparable` object, a value that is an instance of a `Writable` object, an output object, and a reporter. For now, just remember that the reporter is useful. Reporters are discussed in more detail in the "Creating a Custom Mapper and Reducer" section later in this chapter.

■ **Note** The code for the `Mapper.java` and `Reducer.java` interfaces is available from this book's details page at the Apress web site (`http://www.apress.com`), along with the rest of the downloadable code for this book.

The framework will make one call to your map function for each record in your input. There will be multiple instances of your map function running, potentially in multiple Java Virtual Machines (JVMs), and potentially on multiple machines. The framework coordinates all of this for you.

COMMON MAPPERS

One common mapper drops the values and passes only the keys forward:

```
public void map(K key,
                V val,
                OutputCollector<K, V> output,
                Reporter reporter)
    throws IOException {

    output.collect(key, null); /** Note, no value, just a null */

  }
```

Another common mapper converts the key to lowercase:

```
/** put the keys in lower case. */
public void map(Text key,
                V val,
                OutputCollector<Text, V> output,
                Reporter reporter)
    throws IOException {

     Text lowerCaseKey = new Text( key.toString().toLowerCase());
     output.collect(lowerCaseKey, value);

  }
```

A Simple Reduce Function: IdentityReducer

The Hadoop framework calls the reduce function one time for each unique key. The framework provides the key and the set of values that share that key.

The framework-supplied class `IdentityReducer` is a simple example that produces one output record for every value. Listing 2-3 shows this class.

Listing 2-3. *IdentityReducer.java*

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.hadoop.mapred.lib;

import java.io.IOException;

import java.util.Iterator;

import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.MapReduceBase;

/** Performs no reduction, writing all input values directly to the output. */
public class IdentityReducer<K, V>
    extends MapReduceBase implements Reducer<K, V, K, V> {
```

```
/** Writes all keys and values directly to output. */
public void reduce(K key, Iterator<V> values,
                   OutputCollector<K, V> output, Reporter reporter)
  throws IOException {
  while (values.hasNext()) {
    output.collect(key, values.next());
  }
}
```

If you require the output of your job to be sorted, the reducer function must pass the key objects to the `output.collect()` method unchanged. The reduce phase is, however, free to output any number of records, including zero records, with the same key and different values. This particular constraint is also why the map tasks may be multithreaded, while the reduce tasks are explicitly only single-threaded.

COMMON REDUCERS

A common reducer drops the values and passes only the keys forward:

```
public void map(K key,
                V val,
                OutputCollector<K, V> output,
                Reporter reporter)
    throws IOException {

    output.collect(key, null);

  }
```

Another common reducer provides count information for each key:

```
  protected Text count = new Text();
  /** Writes all keys and values directly to output. */
  public void reduce(K key, Iterator<V> values,
                     OutputCollector<K, V> output, Reporter reporter)
    throws IOException {
    int i = 0;
    while (values.hasNext()) {
        i++
    }
    count.set( "" + i );
    output.collect(key, count);
  }
```

Configuring a Job

All Hadoop jobs have a driver program that configures the actual MapReduce job and submits it to the Hadoop framework. This configuration is handled through the `JobConf` object. The sample class `MapReduceIntro` provides a walk-through for using the `JobConf` object to configure and submit a job to the Hadoop framework for execution. The code relies on a class called `MapReduceIntroConfig`, shown in Listing 2-4, which ensures that the input and output directories are set up and ready.

Listing 2-4. *MapReduceIntroConfig.java*

```
package com.apress.hadoopbook.examples.ch2;

import java.io.IOException;
import java.util.Formatter;
import java.util.Random;

import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapred.JobConf;
import org.apache.log4j.Logger;

/** A simple class to handle the housekeeping for the MapReduceIntro
 * example job.
 *
 *
 * <p>
 * This job explicitly configures the job to run, locally and without a
 * distributed file system, as a stand alone application.
 * </p>
 * <p>
 * The input is read from the directory /tmp/MapReduceIntroInput and
 * the output is written to the directory
 * /tmp/MapReduceIntroOutput. If the directory
 * /tmp/MapReduceIntroInput is missing or empty, it is created and
 * some input data files generated. If the directory
 * /tmp/MapReduceIntroOutput is present, it is removed.
 * </p>
 *
 * @author Jason Venner
 */
```

```
public class MapReduceIntroConfig {
    /**
     * Log4j is the recommended way to provide textual information to the user
     * about the job.
     */
    protected static Logger logger =
        Logger.getLogger(MapReduceIntroConfig.class);

    /** Some simple defaults for the job input and job output. */
    /**
     * This is the directory that the framework will look for input files in.
     * The search is recursive if the entry is a directory.
     */
    protected static Path inputDirectory =
        new Path("file:///tmp/MapReduceIntroInput");
    /**
     * This is the directory that the job output will be written to. It must not
     * exist at Job Submission time.
     */
    protected static Path outputDirectory =
        new Path("file:///tmp/MapReduceIntroOutput");

    /**
     * Ensure that there is some input in the <code>inputDirectory</code>,
     * the <code>outputDirectory</code> does not exist and that this job will
     * be run as a local stand alone application.
     *
     * @param conf
     *            The {@link JobConf} object that is required for doing file
     *            system access.
     * @param inputDirectory
     *            The directory the input will reside in.
     * @param outputDirectory
     *            The directory that the output will reside in
     * @throws IOException
     */
    protected static void exampleHouseKeeping(final JobConf conf,
            final Path inputDirectory, final Path outputDirectory)
            throws IOException {
        /**
         * Ensure that this job will be run stand alone rather than relying on
         * the services of an external JobTracker.
         */
        conf.set("mapred.job.tracker", "local");
```

```
        /** Ensure that no global file system is required to run this job. */
        conf.set("fs.default.name", "file:///");
        /**
         * Reduce the in ram sort space, so that the user does not need to
         * increase the jvm memory size. This sets the sort space to 1 Mbyte,
         * which is very small for a real job.
         */
        conf.setInt("io.sort.mb", 1);
        /**
         * Generate some sample input if the <code>inputDirectory</code> is
         * empty or absent.
         */
        generateSampleInputIf(conf, inputDirectory);

        /**
         * Remove the file system item at <code>outputDirectory</code> if it
         * exists.
         */
        if (!removeIf(conf, outputDirectory)) {
            logger.error("Unable to remove " + outputDirectory + "job aborted");
            System.exit(1);
        }
    }

    /**
     * Generate <code>fileCount</code> files in the directory
     * <code>inputDirectory</code>, where the individual lines of the file
     * are a random integer TAB file name.
     *
     * The file names will be file-N where N is between 0 and
     * <code>fileCount</code> - 1. There will be between 1 and
     * <code>maxLines</code> + 1 lines in each file.
     *
     * @param fs
     *            The file system that <code>inputDirectory</code> exists in.
     * @param inputDirectory
     *            The directory to create the files in. This directory must
     *            already exist.
     * @param fileCount
     *            The number of files to create.
     * @param maxLines
     *            The maximum number of lines to write to the file.
     * @throws IOException
     */
```

```
protected static void generateRandomFiles(final FileSystem fs,
        final Path inputDirectory, final int fileCount, final int maxLines)
        throws IOException {

    final Random random = new Random();
    logger .info( "Generating 3 input files of random data," +
                  "each record is a random number TAB the input file name");

    for (int file = 0; file < fileCount; file++) {

        final Path outputFile = new Path(inputDirectory, "file-" + file);
        final String qualifiedOutputFile = outputFile.makeQualified(fs)
                .toUri().toASCIIString();
        FSDataOutputStream out = null;
        try {
            /**
             * This is the standard way to create a file using the Hadoop
             * Framework. An error will be thrown if the file already
             * exists.
             */
            out = fs.create(outputFile);

            final Formatter fmt = new Formatter(out);
            final int lineCount = (int) (Math.abs(random.nextFloat())
                    * maxLines + 1);
            for (int line = 0; line < lineCount; line++) {
                fmt.format("%d\t%s%n", Math.abs(random.nextInt()),
                        qualifiedOutputFile);
            }
            fmt.flush();
        } finally {
            /**
             * It is very important to ensure that file descriptors are
             * closed. The distributed file system code can run out of file
             * descriptors and the errors generated in that case are
             * misleading.
             */
            out.close();
        }
    }
}
```

```
/**
 * This method will generate some sample input, if the
 * <code>inputDirectory</code> is missing or empty.
 *
 * This method also demonstrates some of the basic APIs for interacting
 * with file systems and files. Note: the code has no particular knowledge
 * of the type of file system.
 *
 * @param conf
 *            The Job Configuration object, used for acquiring the
 *            {@link FileSystem} objects.
 * @param inputDirectory
 *            The directory to ensure has sample files.
 * @throws IOException
 */
protected static void generateSampleInputIf(final JobConf conf,
        final Path inputDirectory) throws IOException {

    boolean inputDirectoryExists;
    final FileSystem fs = inputDirectory.getFileSystem(conf);

    if ((inputDirectoryExists = fs.exists(inputDirectory))
            && !isEmptyDirectory(fs, inputDirectory)) {
        if (logger.isDebugEnabled()) {
            logger
                    .debug("The inputDirectory "
                           + inputDirectory
                           + " exists and is either a"
                           + " file or a non empty directory");
        }
        return;
    }

    /**
     * We should only get here if <code>inputDirectory</code> does not
     * exist, or is an empty directory.
     */
    if (!inputDirectoryExists) {
        if (!fs.mkdirs(inputDirectory)) {
            logger.error("Unable to make the inputDirectory "
                    + inputDirectory.makeQualified(fs) + " aborting job");
            System.exit(1);
        }
    }
```

```
    final int fileCount = 3;
    final int maxLines = 100;
    generateRandomFiles(fs, inputDirectory, fileCount, maxLines);
}

/**
 * bean access getter to the {@link #inputDirectory} field.
 *
 * @return the value of inputDirectory.
 */
public static Path getInputDirectory() {
    return inputDirectory;
}

/**
 * bean access getter to the {@link outputDirectory} field.
 *
 * @return the value of outputDirectory.
 */
public static Path getOutputDirectory() {
    return outputDirectory;
}

/**
 * Determine if a directory has any non zero files in it or its descendant
 * directories.
 *
 * @param fs
 *            The {@link FileSystem} object to use for access.
 * @param inputDirectory
 *            The root of the directory tree to search
 * @return true if the directory is missing or does not contain at least one
 *         non empty file.
 * @throws IOException
 */
private static boolean isEmptyDirectory(final FileSystem fs,
        final Path inputDirectory) throws IOException {

    /**
     * This is the standard way to read a directory's contents. This can be
     * quite expensive for a large directory.
     */
    final FileStatus[] statai = fs.listStatus(inputDirectory);
```

```
        /**
         * This method returns null under some circumstances, in particular if
         * the directory does not exist.
         */
        if ((statai == null) || (statai.length == 0)) {
            if (logger.isDebugEnabled()) {
                logger.debug(inputDirectory.makeQualified(fs).toUri()
                        + " is empty or missing");
            }
            return true;
        }
        if (logger.isDebugEnabled()) {
            logger.debug(inputDirectory.makeQualified(fs).toUri()
                    + " is not empty");
        }
        /** Try to find a file in the top level that is not empty. */
        for (final FileStatus status : statai) {
            if (!status.isDir() && (status.getLen() != 0)) {
                if (logger.isDebugEnabled()) {
                    logger.debug("A non empty file "
                            + status.getPath().makeQualified(fs).toUri()
                            + " was found");
                    return false;
                }
            }
        }
        /** Recurse if there are sub directories,
         * looking for a non empty file.
         */
        for (final FileStatus status : statai) {
            if (status.isDir() && isEmptyDirectory(fs, status.getPath())) {
                continue;
            }
            /**
             * If status is a directory it must not be empty or the previous
             * test block would have triggered.
             */
            if (status.isDir()) {
                return false;
            }
        }
        /**
         * Only get here if no non empty files were found in the entire subtree
         * of <code>inputPath</code>.
         */
        return true;
    }
```

```
/**
 * Ensure that the <code>outputDirectory</code> does not exist.
 *
 * <p>
 * The framework requires that the output directory not be present at job
 * submission time.
 * </p>
 * <p>
 * This method also demonstrates how to remove a directory using the
 * {@link FileSystem} API.
 * </p>
 *
 * @param conf
 *            The configuration object. This is needed to know what file
 *            systems and file system plugins are being used.
 * @param outputDirectory
 *            The directory that must be removed if present.
 * @return true if the the <code>outputPath</code> is now missing, or
 *         false if the <code>outputPath</code> is present and was unable
 *         to be removed.
 * @throws IOException
 *             If there is an error loading or configuring the FileSystem
 *             plugin, or other IO error when attempting to access or remove
 *             the <code>outputDirectory</code>.
 */
protected static boolean removeIf(final JobConf conf,
        final Path outputDirectory) throws IOException {

    /** This is standard way to acquire a FileSystem object. */
    final FileSystem fs = outputDirectory.getFileSystem(conf);

    /**
     * If the <code>outputDirectory</code> does not exist this method is
     * done.
     */
    if (!fs.exists(outputDirectory)) {
        if (logger.isDebugEnabled()) {
            logger .debug("The output directory does not exist,"
                          + " no removal needed.");
        }
        return true;
    }
    /**
     * The getFileStatus command will throw an IOException if the path does
     * not exist.
     */
```

```
        final FileStatus status = fs.getFileStatus(outputDirectory);
        logger.info("The job output directory "
                + outputDirectory.makeQualified(fs) + " exists"
                + (status.isDir() ? " and is not a directory" : "")
                + " and will be removed");

        /**
         * Attempt to delete the file or directory. delete recursively just in
         * case <code>outputDirectory</code> is a directory with
         * sub-directories.
         */
        if (!fs.delete(outputDirectory, true)) {
            logger.error("Unable to delete the configured output directory "
                    + outputDirectory);
            return false;
        }

        /** The outputDirectory did exist, but has now been removed. */
        return true;

    }

    /**
     * bean access setter to the {@link inputDirectory} field.
     *
     * @param inputDirectory
     *            The value to set inputDirectory to.
     */
    public static void setInputDirectory(final Path inputDirectory) {
        MapReduceIntroConfig.inputDirectory = inputDirectory;
    }

    /**
     * bean access setter for the {@link outpuDirectory field.
     *
     * @param outputDirectory
     *            The value to set outputDirectory to.
     */
    public static void setOutputDirectory(final Path outputDirectory) {
        MapReduceIntroConfig.outputDirectory = outputDirectory;
    }


}
```

First, you must create a `JobConf` object. It is good practice to pass in a class that is contained in the JAR file that has your map and reduce functions. This ensures that the framework will make the JAR available to the map and reduce tasks run for your job.

```
JobConf conf = new JobConf(MapReduceIntro.class);
```

Now that you have a `JobConfig` object, `conf`, you need to set the required parameters for the job. These include the input and output directory locations, the format of the input and output, and the mapper and reducer classes.

All jobs will have a map phase, and the map phase is responsible for handling the job input. The configuration of the map phase requires you to specify the input locations and the class that will produce the key/value pairs from the input, the mapper class, and potentially, the suggested number of map tasks, map output types, and per-map task threading, as listed in Table 2-2.

Table 2-2. *Map Phase Configuration*

Element	Required?	Default
Input path(s)	Yes	
Class to read and convert the input path elements to key/value pairs	Yes	
Map output key class	No	Job output key class
Map output value class	No	Job output value class
Class supplying the map function	Yes	
Suggested minimum number of map tasks	No	Cluster default
Number of threads to run in each map task	No	1

Most Hadoop Core jobs have their input as some set of files, and these files are either a textual key/value pair per line or a Hadoop-specific binary file format that provides serialized key/value pairs. The class that handles the key/value text input is `KeyValueTextInputFormat`. The class that handles the Hadoop-specific binary file is `SequenceFileInputFormat`.

Specifying Input Formats

The Hadoop framework provides a large variety of input formats. The major distinctions are between textual input formats and binary input formats. The following are the available formats:

- `KeyValueTextInputFormat`: Key/value pairs, one per line.
- `TextInputFormant`: The key is the line number, and the value is the line.
- `NLineInputFormat`: Similar to `KeyValueTextInputFormat`, but the splits are based on *N* lines of input rather than *Y* bytes of input.
- `MultiFileInputFormat`: An abstract class that lets the user implement an input format that aggregates multiple files into one split.
- `SequenceFIleInputFormat`: The input file is a Hadoop sequence file, containing serialized key/value pairs.

`KeyValueTextInputFormat` and `SequenceFileInputFormat` are the most commonly used input formats. The examples in this chapter use `KeyValueTextInputFormat`, as the input files are human-readable.

The following block of code informs the framework of the type and location of the job input:

```
/**
 * This section is the actual job configuration portion /**
 * Configure the inputDirectory and the type of input. In this case
 * we are stating that the input is text, and each record is a
 * single line, and the first TAB is the separator between the key
 * and the value of the record.
 */
conf.setInputFormat(KeyValueTextInputFormat.class);
FileInputFormat.setInputPaths(conf,
                              MapReduceIntroConfig.getInputDirectory());
```

The line `conf.setInputFormat(KeyValueTextInputFormat.class)` informs the framework that all of the files used for input will be textual key/value pairs, one per line.

THE KEYVALUETEXTINPUTFORMAT CLASS

The `KeyValueTextInputFormat` format reads a text file and splits it into records, one record per line. The records are further divided into key/value pairs by splitting the line at the first tab character. If there is no tab character in the line, the entire line is the key, and the value object will contain a zero-length string. There is no way to distinguish an input line that contains a single tab as the last character and the same line without a trailing tab character.

Suppose that an input file has the following three lines, where `TAB` is replaced by an US-ASCII horizontal tab character (0x09):

```
key1TABvalue1
key2
key3TABvalue3TABvalue4
```

Your mapper would be called with the following key/value pairs:

- `key1`, `value1`
- `key2`
- `key3`, `value3TABvalue4`

The actual order in which the keys are passed to your map function is indeterminate. In a real-world example, the actual machine that ran the map that got a given key would be indeterminate. It is very likely, however, that sets of contiguous records in the input will be processed by the same map task, as each task is given one input split from which to work.

The input bytes are considered to be in the UTF-8 character set. As of Hadoop 0.18.2, there is no configurable way to change the character set interpretation of the input files handled by the `KeyValueTextInputFormat` class.

Now that the framework knows where to look for the input files and the class to use to generate key/value pairs from the input files, you need to inform the framework which map function to use.

```
/** Inform the framework that the mapper class will be the {@link
 * IdentityMapper}. This class simply passes the input key-value
 * pairs directly to its output, which in our case will be the
 * shuffle.
 */
conf.setMapperClass(IdentityMapper.class);
```

■ **Note** The simple example in this chapter does not use the optional configuration parameters. If the map function needs to output a different key or value class than the job output, those classes may be set here. In addition, Hadoop supports threading for map functions. This is ideal if the map function is not able to fully utilize the resources allocated for the map task. A simple case of where this might be beneficial is a map task that performs DNS lookups on the IP addresses in a server log.

Setting the Output Parameters

The framework requires that the output parameters be configured, even if the job will not produce any output. The framework will collect the output from the specified tasks (either the output of the map tasks for a MapReduce job that did not include reduce tasks or the output of the job's reduce tasks) and place them into the configured output directory. To avoid issues with file name collisions when placing the task output into the output directory, the framework requires that the output directory not exist when you start the job.

In our simple example, the `MapReduceIntroConfig` class handles ensuring that the output directory does not exist and provides the path to the output directory. The output parameters are actually a little more comprehensive than just the setting of the output path. The code will also set the output format and the output key and value classes.

The `Text` class is the functional equivalent of a `String`. It implements the `WritableComparable` interface, which is necessary for keys, and the `Writable` interface (which is actually a subset of `WritableComparable`), which is necessary for values. Unlike `String`, `Text` is mutable, and the `Text` class has some explicit methods for UTF-8 byte handling.

The key feature of a `Writable` is that the framework knows how to serialize and deserialize a `Writable` object. The `WritableComparable` adds the `compareTo` interface so the framework knows how to sort the `WritableComparable` objects. The interface references for `WritableComparable` and `Writable` are shown in Listings 2-5 and 2-6.

The following code block provides an example of the minimum required configuration for the output of a MapReduce job:

```
/** Configure the output of the job to go to the output directory.
 * Inform the framework that the Output Key and Value classes will be
 * {@link Text} and the output file format will {@link
 * TextOutputFormat}. The TextOutput format class produces a record of
 * output for each Key,Value pair, with the following format.
 * Formatter.format( "%s\t%s%n", key.toString(), value.toString() );.
 *
 * In addition indicate to the framework that there will be
 * 1 reduce. This results in all input keys being placed
 * into the same, single, partition, and the final output
 * being a single sorted file.
 */
FileOutputFormat.setOutputPath(conf,
                               MapReduceIntroConfig.getOutputDirectory());
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
```

The `FileOutputFormat.setOutputPath(conf, MapReduceIntroConfig.getOutputDirectory())` setting is familiar from the input example discussed earlier in the chapter. The `conf.setOutputKeyClass(Text.class)` and `conf.setOutputValueClass(Text.class)` settings are new. These settings inform the framework of the types of the key/value pairs to expect for the reduce phase. By default, these classes will also be used to set the values the framework will expect from the map output. Unsurprisingly, the method to set the output key class for the map output is `conf.setMapOutputKeyClass(Class<? extends WritableComparable>)`. To set the output value class, the method is `conf.setMapOutputValueClass(Class<? extends Writable>)`.

Listing 2-5. *WritableComparable.java*

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

```
package org.apache.hadoop.io;

/**
 * A {@link Writable} which is also {@link Comparable}.
 *
 * <p><code>WritableComparable</code>s can be compared to each other, typically
 * via <code>Comparator</code>s. Any type which is to be used as a
 * <code>key</code> in the Hadoop Map-Reduce framework should implement this
 * interface.</p>
 *
 * <p>Example:</p>
 * <p><blockquote><pre>
 *     public class MyWritableComparable implements WritableComparable {
 *       // Some data
 *       private int counter;
 *       private long timestamp;
 *
 *       public void write(DataOutput out) throws IOException {
 *         out.writeInt(counter);
 *         out.writeLong(timestamp);
 *       }
 *
 *       public void readFields(DataInput in) throws IOException {
 *         counter = in.readInt();
 *         timestamp = in.readLong();
 *       }
 *
 *       public int compareTo(MyWritableComparable w) {
 *         int thisValue = this.value;
 *         int thatValue = ((IntWritable)o).value;
 *         return (thisValue &lt; thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
 *       }
 *     }
 * </pre></blockquote></p>
 */
public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

Listing 2-6. *Writable.java*

```
/**
 * Licensed to the Apache Software Foundation (ASF) under one
 * or more contributor license agreements.  See the NOTICE file
 * distributed with this work for additional information
 * regarding copyright ownership.  The ASF licenses this file
```

```
 * to you under the Apache License, Version 2.0 (the
 * "License"); you may not use this file except in compliance
 * with the License.  You may obtain a copy of the License at
 *
 *     http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

/**
 * A serializable object which implements a simple, efficient, serialization
 * protocol, based on {@link DataInput} and {@link DataOutput}.
 *
 * <p>Any <code>key</code> or <code>value</code> type in the Hadoop Map-Reduce
 * framework implements this interface.</p>
 *
 * <p>Implementations typically implement a static <code>read(DataInput)</code>
 * method which constructs a new instance, calls {@link #readFields(DataInput)}
 * and returns the instance.</p>
 *
 * <p>Example:</p>
 * <p><blockquote><pre>
 *     public class MyWritable implements Writable {
 *       // Some data
 *       private int counter;
 *       private long timestamp;
 *
 *       public void write(DataOutput out) throws IOException {
 *         out.writeInt(counter);
 *         out.writeLong(timestamp);
 *       }
 *
 *       public void readFields(DataInput in) throws IOException {
 *         counter = in.readInt();
 *         timestamp = in.readLong();
 *       }
 *
```

```
 *       public static MyWritable read(DataInput in) throws IOException {
 *         MyWritable w = new MyWritable();
 *         w.readFields(in);
 *         return w;
 *       }
 *     }
 * </pre></blockquote></p>
 */
public interface Writable {
  /**
   * Serialize the fields of this object to <code>out</code>.
   *
   * @param out <code>DataOuput</code> to serialize this object into.
   * @throws IOException
   */
  void write(DataOutput out) throws IOException;

  /**
   * Deserialize the fields of this object from <code>in</code>.
   *
   * <p>For efficiency, implementations should attempt to re-use storage in the
   * existing object where possible.</p>
   *
   * @param in <code>DataInput</code> to deseriablize this object from.
   * @throws IOException
   */
  void readFields(DataInput in) throws IOException;
}
```

Configuring the Reduce Phase

To configure the reduce phase, the user must supply the framework with five pieces of information:

- The number of reduce tasks; if zero, no reduce phase is run
- The class supplying the reduce method
- The input key and value types for the reduce task; by default, the same as the reduce output
- The output key and value types for the reduce task
- The output file type for the reduce task output

The input and output key and value types, as well as the output file type, are the same as those covered in the previous “Setting the Output Parameters” section. Here, we will look at setting the number of reduce tasks and the reducer class.

The configured number of reduce tasks determines the number of output files for a job that will run the reduce phase. Tuning this value will have a significant impact on the overall performance of your job. The time spent sorting the keys for each output file is a function of the number of keys. In addition, the number of reduce tasks determines the maximum number of reduce tasks that can be run in parallel.

The framework generally has a default number of reduce tasks configured. This value is set by the `mapred.reduce.tasks` parameter, which defaults to 1. This will result in a single output file containing all of the output keys, in sorted order. There will be one reduce task, run on a single machine that processes every key.

The number of reduce tasks is commonly set in the configuration phase of a job.

```
conf.setNumReduceTasks(1);
```

In general, unless there is a significant need for a single output file, the number of reduce tasks is set to roughly the number of simultaneous execution slots in the cluster. In Chapter 9, the class `DataJoinReduceOutput` is provided as a sample for efficiently merging multiple reduce task outputs into a single sorted file.

CLUSTER EXECUTION SLOTS

A typical cluster is composed of *M* TaskTracker machines, with *C* CPUs, each of which supports *T* threads. This would result in *M* * *C* * *T* execution slots in the cluster. In my environment, the machines typically have eight CPUs that support one thread per CPU, and a small cluster might have ten TaskTracker machines. This gives us 10 * 8 * 1 = 80 execution slots in the cluster.

If your tasks tend not to be CPU-bound, you may adjust the number of execution slots configured to optimize the CPU utilization on your TaskTracker machines.

The configuration parameter `mapred.tasktracker.map.tasks.maximum` controls the maximum number of map tasks that will be run simultaneously on a TaskTracker node.

The configuration parameter `mapred.tasktracker.reduce.tasks.maximum` controls the maximum number of reduce tasks that will be run simultaneously on a TaskTracker node.

This requires tuning on a per-job basis and is a weakness in Hadoop at present, as the maximum values are not per-job configurable and instead require a cluster restart.

The reducer class needs to be set only if the number of reduce tasks is not zero. It is very common to not need a reducer, since frequently you do not require sorted output or value grouping by key. The actual setting of the reducer class is straightforward:

```
/** Inform the framework that the reducer class will be the
 * {@link IdentityReducer}. This class simply writes an output record
 * key/value record for each value in the key/value set it receives as
 * input. The value ordering is arbitrary.
 */
conf.setReducerClass(IdentityReducer.class);
```

A COMMON EXCEPTION

The framework relies on the output parameters being set correctly. One of the more common errors is to have each reduce task fail with an exception of the form:

```
java.io.IOException: Type mismatch in key from map: expected
org.apache.hadoop.io.LongWritable, recieved org.apache.hadoop.io.Text
```

This error indicates that output key class has been defaulted by the framework, or was set incorrectly during the job configuration.

To correct this, use the following:

```
conf.setOutputKeyClass( Text.class )
```

Or if your map output is not the same as your job output, use this form:

```
conf.setMapOutputKeyClass( Text.class )
```

This error may occur for the value class as well:

```
java.io.IOException: Type mismatch in value from map: expected
org.apache.hadoop.io.LongWritable, recieved org.apache.hadoop.io.Text
```

The corresponding `setOutputValueClass()` or `setMapOutputValue()` class methods are needed to correct this.

Running a Job

The ultimate aim of all your MapReduce job configuration is to actually run that job. The `MapReduceIntro.java` example (Listing 2-1) demonstrates a common and simple way to run a job:

```
logger .info("Launching the job.");
/** Send the job configuration to the framework
 * and request that the job be run.
*/
final RunningJob job = JobClient.runJob(conf);
logger.info("The job has completed.");
```

The method `runJob()` submits the configuration information to the framework and waits for the framework to finish running the job. The response is provided in the `job` object.

The `RunningJob` class provides a number of methods for examining the response. Perhaps the most useful is `job.isSuccessful()`.

Run `MapReduceIntro.java` as follows (using the `CH2.jar` file provided with this book's downloadable code):

```
hadoop jar DOWNLOAD_PATH/ch2.jar ➥
com.apress.hadoopbook.examples.ch2.MapReduceIntro
```

The response should be as follows:

```
ch2.MapReduceIntroConfig: Generating 3 input files of random data, each record
is a random number TAB the input file name
ch2.MapReduceIntro: Launching the job.
jvm.JvmMetrics: Initializing JVM Metrics with processName=JobTracker, sessionId=
mapred.JobClient: Use GenericOptionsParser for parsing the arguments.
Applications should implement Tool for the same.
mapred.FileInputFormat: Total input paths to process : 3
mapred.FileInputFormat: Total input paths to process : 3
mapred.FileInputFormat: Total input paths to process : 3
mapred.FileInputFormat: Total input paths to process : 3
mapred.JobClient: Running job: job_local_0001
mapred.MapTask: numReduceTasks: 1
mapred.MapTask: io.sort.mb = 1
mapred.MapTask: data buffer = 796928/996160
mapred.MapTask: record buffer = 2620/3276
mapred.MapTask: Starting flush of map output
mapred.MapTask: bufstart = 0; bufend = 664; bufvoid = 996160
mapred.MapTask: kvstart = 0; kvend = 14; length = 3276
mapred.MapTask: Index: (0, 694, 694)
mapred.MapTask: Finished spill 0
mapred.LocalJobRunner: file:/tmp/MapReduceIntroInput/file-2:0+664
mapred.TaskRunner: Task 'attempt_local_0001_m_000000_0' done.
mapred.TaskRunner: Saved output of task 'attempt_local_0001_m_000000_0' to
file:/tmp/MapReduceIntroOutput
mapred.MapTask: numReduceTasks: 1
mapred.MapTask: io.sort.mb = 1
mapred.MapTask: data buffer = 796928/996160
mapred.MapTask: record buffer = 2620/3276
mapred.MapTask: Starting flush of map output
mapred.MapTask: bufstart = 0; bufend = 3418; bufvoid = 996160
mapred.MapTask: kvstart = 0; kvend = 72; length = 3276
mapred.MapTask: Index: (0, 3564, 3564)
mapred.MapTask: Finished spill 0
mapred.LocalJobRunner: file:/tmp/MapReduceIntroInput/file-1:0+3418
mapred.TaskRunner: Task 'attempt_local_0001_m_000001_0' done.
mapred.TaskRunner: Saved output of task 'attempt_local_0001_m_000001_0' to
file:/tmp/MapReduceIntroOutput
mapred.MapTask: numReduceTasks: 1
mapred.MapTask: io.sort.mb = 1
mapred.MapTask: data buffer = 796928/996160
mapred.MapTask: record buffer = 2620/3276
```

```
mapred.MapTask: Starting flush of map output
mapred.MapTask: bufstart = 0; bufend = 3986; bufvoid = 996160
mapred.MapTask: kvstart = 0; kvend = 84; length = 3276
mapred.MapTask: Index: (0, 4156, 4156)
mapred.MapTask: Finished spill 0
mapred.LocalJobRunner: file:/tmp/MapReduceIntroInput/file-0:0+3986
mapred.TaskRunner: Task 'attempt_local_0001_m_000002_0' done.
mapred.TaskRunner: Saved output of task 'attempt_local_0001_m_000002_0' to
file:/tmp/MapReduceIntroOutput
mapred.ReduceTask: Initiating final on-disk merge with 3 files
mapred.Merger: Merging 3 sorted segments
mapred.Merger: Down to the last merge-pass, with 3 segments left of total size:
8414 bytes
mapred.LocalJobRunner: reduce > reduce
mapred.TaskRunner: Task 'attempt_local_0001_r_000000_0' done.
mapred.TaskRunner: Saved output of task 'attempt_local_0001_r_000000_0' to
file:/tmp/MapReduceIntroOutput
mapred.JobClient: Job complete: job_local_0001
mapred.JobClient: Counters: 11
mapred.JobClient:   File Systems
mapred.JobClient:     Local bytes read=230060
mapred.JobClient:     Local bytes written=319797
mapred.JobClient:   Map-Reduce Framework
mapred.JobClient:     Reduce input groups=170
mapred.JobClient:     Combine output records=0
mapred.JobClient:     Map input records=170
mapred.JobClient:     Reduce output records=170
mapred.JobClient:     Map output bytes=8068
mapred.JobClient:     Map input bytes=8068
mapred.JobClient:     Combine input records=0
mapred.JobClient:     Map output records=170
mapred.JobClient:     Reduce input records=170
ch2.MapReduceIntro: The job has completed.
ch2.MapReduceIntro: The job completed successfully.
```

Congratulations, you have run a MapReduce job.

The single output file of the reduce task in the file `/tmp/MapReduceIntroOutput/part-00000` will have a series of lines of the form *`Number`* `TAB file:/tmp/MapReduceIntroInput/file-`*`N`*. The first thing you will notice is that the numbers don't seem to be in order. The code that generates the input produces a random number for the key of each line, but the example tells the framework that the keys are `Text`. Therefore, the numbers have been sorted as text rather than as numbers.

Creating a Custom Mapper and Reducer

As you've seen, your first Hadoop job, in `MapReduceIntro`, produced sorted output, but the sorting was not suitable, as it sorted lexically rather than numerically, and the keys for the job were numbers. Now, let's work out what is required to sort numerically, using a custom mapper. Then we'll look at a custom reducer that outputs the values in a format that is easy to parse.

Setting Up a Custom Mapper

Sorting numerically doesn't sound difficult. Let's try making the output key class a `LongWritable`, another class supplied by the framework:

```
conf.setOutputKeyClass(LongWritable.class);
```

instead of:

```
conf.setOutputKeyClass(Text.class);
```

The class with this change is available as `MapReduceIntroLongWritable.java`. Run this class via this command:

```
hadoop jar DOWNLOAD_PATH/ch2.jar ➥
com.apress.hadoopbook.examples.ch2.MapReduceIntroLongWritable
```

You will see the following in the output:

```
mapred.LocalJobRunner: job_local_0001
java.io.IOException: Type mismatch in key from map: expected
org.apache.hadoop.io.LongWritable, recieved org.apache.hadoop.io.Text
    at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.collect(MapTask.java:415)
    at org.apache.hadoop.mapred.lib.IdentityMapper.map(IdentityMapper.java:37)
    at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:47)
    at org.apache.hadoop.mapred.MapTask.run(MapTask.java:227)
    at org.apache.hadoop.mapred.LocalJobRunner$Job.run(LocalJobRunner.java:157)
ch2.MapReduceIntroLongWritable: The job has failed due to an IO Exception
```

As you can see, just changing the output key class was insufficient. If you are going to change the output key class to a `LongWritable`, you also need to modify the map function so that it outputs `LongWritable` keys.

For the job to actually produce output that is sorted numerically, you must change the job configuration and provide a custom mapper class. This is done by two calls on the `JobConf` object:

- `conf.setOutputKeyClass(LongWritable.class)`: Informs the framework of the key class for map and reduce output.
- `conf.setMapperClass(TransformKeysToLongMapper.class)`: Informs the framework of the custom class that provides the map method that takes as input `Text` keys and outputs `LongWritable` keys.

A demonstration class `MapReduceIntroLongWritableCorrect.java` provides the configuration for this. This class is identical to `MapReduceIntro`, except for these two replacement method calls.

■ **Note** The job configuration could also provide a custom sort option. One way to do this is to provide a custom class that implements `WritableComparable` and use that as the key class. Another way is to specify a `CustomComparator` in the job configuration via the `setOutputKeyComparatorClass()` method on the `JobConf` object. An example of implementing a custom comparator is provided in Chapter 9.

You also need to provide a mapper class that performs the transformation. The sample mapper class `TransformKeysToLongMapper.java` does this. The `TransformKeysToLongMapper.java` class file has a number of changes from the `IdentityMapper` class (shown earlier in Listing 2-2).

First, the class declaration is no longer generic; the types have been made concrete:

```
/** Transform the input Text, Text key value
 * pairs into LongWritable, Text key/value pairs.
*/
public class TransformKeysToLongMapperMapper
  extends MapReduceBase implements Mapper<Text, Text, LongWritable, Text>
```

Notice that the code actually provides the types for the key/value pairs for input and for output. The original `IdentityMapper` class was completely generic. In addition, the identity mapper's declaration was `implements Mapper<K, V, K, V>`. In `TransformKeysToLongMapperMapper`, the declaration is `implements Mapper<Text, Text, LongWritable, Text>`.

The `map()` method of `TransformKeysToLongMapper` is substantially different from the `IdentityMapper` and introduces the use of the reporter object.

The Reporter Object

The map and reduce methods both take four parameters: the key, the value, the output collector, and the reporter. The `reporter` object provides a mechanism for informing the framework of the current status of your job.

The `reporter` object provides three methods:

- `incrCounter()`: Provides counters that are aggregated and reported at the end of the job.
- `setStatus()`: Provides a status line for this map or reduce task.
- `getInputSplit()`: Returns information about the input source for this task. If the input is simple files, this can provide useful information for log messages.

Each call on the `reporter` object or the output collector provides a heartbeat to the framework, informing it that the task is not deadlocked or otherwise unresponsive. If your map or reduce method takes substantial time, the method must make periodic calls on the `reporter`

object methods, to inform the framework that it is still working. The framework will kill tasks that have not reported in 600 seconds by default.

Listing 2-6 shows the body of the `TransformKeysToLongMapper` mapper that uses the `reporter` object.

Listing 2-6. *The Reporter Object in TransformKeysToLongMapper.java*

```
/** Map input to the output, transforming the input {@link Text}
 * keys into {@link LongWritable} keys.
 * The values are passed through unchanged.
 *
 * Report on the status of the job.
 * @param key The input key, supplied by the framework, a {@link Text} value.
 * @param value The input value, supplied by the framework, a {@link Text} value.
 * @param output The {@link OutputCollector} that takes
 * {@link LongWritable}, {@link Text} pairs.
 * @param reporter The object that provides a way
 * to report status back to the framework.
 * @exception IOException if there is any error.
 */
public void map(Text key, Text value,
                OutputCollector<LongWritable, Text> output, Reporter reporter)
  throws IOException {

    try {
        try {
            reporter.incrCounter( "Input", "total records", 1 );
            LongWritable newKey =
                  new LongWritable( Long.parseLong( key.toString() ) );
            reporter.incrCounter( "Input", "parsed records", 1 );
            output.collect(newKey, value);
        } catch( NumberFormatException e ) {
            /** This is a somewhat expected case and we handle it specially. */
            logger.warn( "Unable to parse key as a long for key,"
                              +" value " + key + " " + value, e );
            reporter.incrCounter( "Input", "number format", 1 );
            return;
        }
    } catch( Throwable e ) {
        /** It is very important to report back if there were
          * exceptions in the mapper.
         * In particular it is very handy to report the number of exceptions.
         * If this is done, the driver can make better assumptions
         * on the success or failure of the job.
         */
```

```
            logger.error( "Unexpected exception in mapper for key,"
                 + " value " + key + ", " + value, e );
            reporter.incrCounter( "Input", "Exception", 1 );
            reporter.incrCounter( "Exceptions", e.getClass().getName(), 1 );
            if (e instanceof IOException) {
                throw (IOException) e;
            }
            if (e instanceof RuntimeException) {
                throw (RuntimeException) e;
            }
            throw new IOException( "Unknown Exception", e );
        }
    }
```

This block of code introduces a new object, `reporter`, and some best practice patterns. The key piece of this is the transformation of the `Text` key to a `LongWritable` key.

```
LongWritable newKey = new LongWritable(Long.parseLong(key.toString()));
output.collect(newKey, value);
```

The code in Listing 2-6 is sufficient to perform the transformation, and also includes some additional code for tracking and reporting.

CODE EFFICIENCY

The pattern of creating a new key object in the mapper for the transformation object is not the most efficient pattern. Most key classes provide a `set()` method, which sets the current value of the key. The `output.collect()` method uses the current value of the key, and once the `collect()` method is complete, the key object or the value object is free to be reused.

If the job is configured to multithread the map method, via `conf.setMapRunner(MultithreadedMapRunner.class)`, the map method will be called by multiple threads. Extreme care must be taken in using the mapper class member variables. A `ThreadLocal LongWritable` object could be used to ensure thread safety. To simplify the example, a new `LongWritable` is constructed. In the reduce method; there are no threading issues.

Object churn is a significant performance issue in a map method, and to a lesser extent, in the reduce method. Object reuse can provide a significant performance gain.

The Counters and Exceptions

This example includes two `try/catch` blocks and several calls to the `reporter.incrCounter()` method. It is a good practice to wrap your map and reduce methods in a `try` block that catches `Throwables` and reports on the catches.

The JobTracker, the Hadoop Core server process that manages job execution on the cluster, accumulates the counter values and provides a final count in the job output, as well

as making the instantaneous count available in the JobTracker web interface (available on `http://jobtracker_host:50030/` by default). This interface will be discussed in more detail in Chapter 6, which covers the setup of a multimachine cluster.

You can now run the job:

```
hadoop jar ch2.jar ➥
com.apress.hadoopbook.examples.ch2.MapReduceIntroLongWritableCorrect
```

The output that reflects the counters is as follows:

```
mapred.JobClient: Job complete: job_local_0001
mapred.JobClient: Counters: 13
mapred.JobClient:   File Systems
mapred.JobClient:     Local bytes read=78562
mapred.JobClient:     Local bytes written=157868
mapred.JobClient:   Input
mapred.JobClient:     total records=126
mapred.JobClient:     parsed records=126
mapred.JobClient:   Map-Reduce Framework
mapred.JobClient:     Reduce input groups=126
mapred.JobClient:     Combine output records=0
mapred.JobClient:     Map input records=126
mapred.JobClient:     Reduce output records=126
mapred.JobClient:     Map output bytes=5670
mapred.JobClient:     Map input bytes=5992
mapred.JobClient:     Combine input records=0
mapred.JobClient:     Map output records=126
mapred.JobClient:     Reduce input records=126
```

The first `catch` block handles exceptions related to `reporter.incrCounter( "Input", "number format", 1 );`, which may be thrown during the key transformation:

```
} catch( NumberFormatException e ) {
    /** This is a somewhat expected case and we handle it specially. */
    reporter.incrCounter( "Input", "number format", 1 );
    return;
}
```

You expect that some of the keys may not convert correctly into `Long` values, so you capture the exception. The `reporter.incrCounter()` call tells the framework to increment a counter in the `Input` group, of the name `number format`, by 1. If the counter does not already exist, it will be created.

In the sample input, there are no records that will cause a number format exception. The only counters that are accumulated are `Input.total records` and `Input.parsed records`. These two counters will show up in the job output as part of the `Input` group:

```
mapred.JobClient:   Input
mapred.JobClient:     total records=126
mapred.JobClient:     parsed records=126
```

If one or more keys caused an exception during the conversion to `Long`, the output might look more like this:

```
mapred.JobClient:   Input
mapred.JobClient:     total records=126
mapred.JobClient:     parsed records=125
mapred.JobClient:    number format=1
```

■ **Note** The sum of the parsed records and the number formats should equal the total records. The counters are also available via the `RunningJob` object, allowing for a more comprehensive check of the success status. The totals for your job will vary from this example.

After the Job Finishes

Once the job finishes, the framework will provide you with a filled-out `RunningJob` object. This object has information about the framework's opinion on the success status of your job via the `conf.isSuccessful()` method. The framework will report that the job was unsuccessful if it was unable to complete any single map task or if the job was killed.

This generally doesn't provide enough information to make a determination on the actual success. It may be that there was an exception in the map or method for every key or for most keys. If the map or reduce function provides job counters for these cases, your job driver will be able to make a better determination regarding the actual success or failure of your job.

In the sample mapper, several counters were collected under different circumstances:

- `reporter.incrCounter( TransformKeysToLongMapper.INPUT, TransformKeysToLongMapper.TOTAL_RECORDS, 1 )`: Reports the total number of input records seen.
- `reporter.incrCounter( TransformKeysToLongMapper.INPUT, TransformKeysToLongMapper.PARSED_RECORDS, 1 )`: Reports the total number of records successfully parsed.
- `reporter.incrCounter( TransformKeysToLongMapper.INPUT, TransformKeysToLongMapper.NUMBER_FORMAT, 1 )`: Reports the total number of records where the key could not be parsed.

- `reporter.incrCounter( TransformKeysToLongMapper.INPUT, TransformKeysToLongMapper.EXCEPTION, 1 )`: Reports the number of records that generated an exception when being processed.
- `reporter.incrCounter( TransformKeysToLongMapper.EXCEPTIONS, e.getClass().getName(), 1 )`: Reports the counts of exceptions by type.

Examining the Counters

Once the framework fills in the `RunningJob` object and returns control back to the job driver, the driver is able to examine the values of the various counters, as well as the framework's success or failure status.

Making the counter values available is a multistep process.

```
/** Get the job counters. {@see RunningJob.getCounters()}. */
Counters jobCounters = job.getCounters();

/** Look up the "Input" Group of counters. */
Counters.Group inputGroup = jobCounters.getGroup( TransformKeysToLongMapper.INPUT );

/** The map task potentially outputs 4 counters in the input group.
  * Get each of them.
  */
long total = inputGroup.getCounter( TransformKeysToLongMapper.TOTAL_RECORDS );
long parsed = inputGroup.getCounter( TransformKeysToLongMapper.PARSED_RECORDS );
long format = inputGroup.getCounter( TransformKeysToLongMapper.NUMBER_FORMAT );
long exceptions = inputGroup.getCounter( TransformKeysToLongMapper.EXCEPTION );
```

Now that the job driver has the counters issued by the map method, a much more accurate determination of success can be made.

■ **Caution** An accurate determination of success is critical. In one of my production clusters, a TaskTracker node was incorrectly configured. The result of this misconfiguration was that none of the computationally intense work could be run in the map task, and the map method would return immediately with an exception. As far as the framework was concerned, this machine was super fast, and it scheduled almost all of the map tasks on this machine. The job was successful as far as the framework was concerned, but totally unsuccessful per the business rules. At that point. the pattern of checking the exception count was not part of the standard practice, and the failure was uncovered only when the consumer of the results noticed there were no valid results. Save yourself much embarrassment—collect information about the successes and failures in the mapper and reducer objects and check those results in your job driver.

Was This Job Really Successful?

The check for success primarily involves ensuring that the number of records output is roughly the same as the number of records input. Hadoop jobs are generally dealing with bulk real-world data, which is never 100% clean, so a small error rate is generally acceptable.

```
if (format != 0) {
    logger.warn( "There were " + format + " keys that were not "
                + "transformable to long values");
}

/** Check to see if we had any unexpected exceptions.
  * This usually indicates some significant problem,
  * either with the machine running the task that had
  * the exception, or the map or reduce function code.
  * Log an error for each type of exception with the count.
  */
if (exceptions > 0 ) {
    Counters.Group exceptionGroup = jobCounters.getGroup(
                      TransformKeysToLongMapper.EXCEPTIONS );
    for (Counters.Counter counter : exceptionGroup) {
        logger.error( "There were " + counter.getCounter()
                 + " exceptions of type " + counter.getDisplayName() );
    }
}

if (total == parsed) {
    logger.info("The job completed successfully.");
    System.exit(0);
}

// We had some failures in handling the input records.
// Did enough records process for this to be a successful job?
// is 90% good enough?
if (total * .9 <= parsed) {
    logger.warn( "The job completed with some errors, "
                    + (total - parsed) + " out of " + total );
    System.exit( 0 );
}

logger.error( "The job did not complete successfully,"
    +" too many errors processing the input, only "
    + parsed + " of " + total + "records completed" );
System.exit( 1 );
```

In this particular case, you would expect a small number of `NumberFormatExceptions` but no other exceptions. If the total number of input records is roughly the number of parsed input records, and you have no unexpected exceptions, this job is a success.

Creating a Custom Reducer

The reduce method is called once for each key, and passes the key and an iterator to all of the map output values that share that key. The reduce task is an ideal place for summarizing data and for doing basic duplicate suppression.

■ **Note** For managing duplicate suppression against a prior seen set, it is usually best to keep the prior seen set in either HBase (the Hadoop database) or in a sorted format, such as a Hadoop map file. If this is not done, then the dataset of *seen* records and the dataset of *input* records must be merged and sorted, which can take considerable time if either dataset is large. In the HBase case, if the input data is already sorted, the duplicate status of an input record can be rapidly determined. With a simple sorted seen set, map-side joins may be performed. HBase is discussed in Chapter 10, and map-side joins are covered in Chapters 8 and 9.

For the sample custom reducer, let's merge the values into a comma-separated values (CSV) form, so you have one output line per key, with all of the values in a simple-to-parse format.

After your work with the custom mapper in the preceding sections, creating a custom reducer will seem familiar. This version is in `MapReduceIntroLongWritableReduce.java`, which is based on `MapReduceIntroLongWritableCorrect.java`. First, the framework needs to be informed of the reducer class. The key piece is, as usual, to inform the framework of the reducer class, so add the following single line:

```
/** Inform the framework that the reducer class will be the
  * {@link MergeValuesToCSV}.
 * This class simply writes an output record key,
 * value record for each value in the key, valueset it receives as
 * input.
 * The value ordering is arbitrary.
 */
conf.setReducerClass(MergeValuesToCSV.class);
```

There have been no changes to the output classes, so no other changes are required to `MapReduceIntroLongWritableCorrect.java`.

The class to actually perform the work is `MergeValuesToCSVReducer.java`. As with the mapper example, `TransformKeysToLongMapper`, you start with your class declaration, which has partially specified the generic types:

```
public class MergeValuesToCSVReducer<K, V>
    extends MapReduceBase implements Reducer<K, V, K, Text> {
```

The reduce method doesn't need to know the incoming value class; it requires only the `toString()` method to work. The reduce method does need to construct a new output value, and for simplicity's sake, given this transformation, the output value is declared to be `Text`.

The actual method declaration also has the same type specification:

```
/** Merge the values for each key into a CSV text string.
 *
 * @param key The key object for this group.
 * @param values Iterator to the set of values that share the <code>key</code>.
 * @param output The {@see OutputCollector} to pass the transformed output to.
 * @param reporter The reporter object to update counters and set task status.
 * @exception IOException if there is an error.
 */
```

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

The framework will throw an error if the job is expecting a different output value type than `Text`. As with the mapper example, you have a method body that employs the `reporter.incrCounter()` method to make detailed information available to the job and via the web interface. As a performance optimization, to reduce object churn, two class fields are declared. These variables are used in the `reduce()` method:

```
/** Used to construct the merged value.
  * The {@link Text.set() Text.set} method is used
  * to prevent object churn.
  */
protected Text mergedValue = new Text();
/** Working storage for constructing the resulting string. */
protected StringBuilder buffer = new StringBuilder();
```

The `buffer` object is used to build the CSV-style line for the output, and `mergedValue` is the actual object that is sent to the output on each `reduce()` call. It is safe to declare these as class fields, rather than as local variables, because the individual reduce tasks are run only as single threads by the framework.

■ **Note** There may be multiple reduce tasks running simultaneously, but each task is running in a separate JVM, and the JVMs are potentially running on separate physical machines.

The `reduce()` method is called with the key and an iterator to the values that share that key. Recall that, ideally, a reduce task will make no changes to the key, and will use that key as the key argument to the `output.collect()` method calls in the `reduce()` method. The design goal for this `reduce()` method is to output only a single row for every key, with a comma-separated list of the values that shared that key. The core of the `reduce()` method has a bit of boilerplate for the object churn optimizations to reset the `StringBuilder` object, and a loop to process each of the values for this key:

```
buffer.setLength(0);
for (;values.hasNext(); valueCount++) {
    reporter.incrCounter( OUTPUT, MergeValuesToCSVReducer.TOTAL_VALUES, 1 );
    String value = values.next().toString();
    if (value.contains("\"")) { // Perform Excel style quoting
        value.replaceAll( "\"", "\\\"" );
    }
    buffer.append( '"' );
    buffer.append( value );
    buffer.append( "\"," );
}
buffer.setLength( buffer.length() - 1 );
```

It is rare that a `reduce()` method doesn't have a loop that iterates over the values. It is good form to report on the number of values input. In this example, `reporter.incrCounter ( OUTPUT, MergeValuesToCSVReducer.TOTAL_VALUES, 1 )` handles the reporting.

This reducer relies on the `toString()` method of the value object, which seems reasonable for a textual output job, as the framework would also be using the `toString()` method to produce the output. The rest of the preceding code block simply builds a comma-separated list of values, with Excel-style CSV quoting.

The actual output block must build a new value for the output. In this case, a class field `mergedValue` will be used. In a larger job, there may be a billion keys passed through the `reduce()` method, and by using the class field, the amount of object churn is greatly reduced. In this example, there are also counters for the output records:

```
mergedValue.set(buffer.toString());
reporter.incrCounter( OUTPUT, TOTAL_OUTPUT_RECORDS, 1 );
output.collect( key, mergedValue );
```

The value is set on the `mergedValue` object, using the `mergedValue.set(buffer.toString())` statement, and the value is output using the `output.collect( key, mergedValue )` line. This example uses `Text` as the output value class; it is acceptable to use any `Writable` as the output value class. If the output format is a `SequenceFile`, there is no need for a functional `toString()` method on your object.

■ **Note** The framework serializes the key and value into the output stream during the `collect()` method, leaving the user free to change the objects values when the method returns.

Why Do the Mapper and Reducer Extend MapReduceBase?

The custom mapper class `TransformKeysToLongMapper` and reducer class `MergeValuesToCSVReducer` both extend the class `org.apache.hadoop.mapred.MapReduceBase`. This class provides basic implementations of two additional methods that are required of a mapper or a reducer by the framework. The framework calls the `configure()` method upon initializing a task, and it calls the `close()` method when the task has finished processing its input split:

```
/** Default implementation that does nothing. */
public void close() throws IOException {
}

/** Default implementation that does nothing. */
public void configure(JobConf job) {
}
```

The configure Method

The `configure()` method is the only way to get access to the `JobConf` object for your task. This method is where any per-task configuration and setup is done. If your application relies on

the Spring Framework for setup, the application context would be established here and the relevant beans found.

It is very common for the developer to have a `JobConf` member variable, which would be initialized in this method with the passed-in `JobConf` object. (I prefer to issue a logging record with detailed information about the input split.) The `configure()` method is also the ideal place to open additional files that need to be read or written to during the `map()` or `reduce()` method.

The close Method

The `close()` method is called by the framework when all of the input-split entries have been processed by the applicable `map()` or `reduce()` method. It is very important to close any supplemental files here to ensure that they are properly flushed to the file system. Particularly for HDFS, if the file is not closed, data in the last block may be lost.

The following example also makes a reporter call in the `close()` method:

```
/** Keep track of the maximum number of keys a value had.
 * Report it in the counters so that per task counters can be examined as needed
 * and set the task status to include this maximum count.
 */
@Override
public void close() throws IOException {
    super.close();
    if (reporter!=null) {
        reporter.incrCounter( OUTPUT, MAX_VALUES, maxValueCount );
        reporter.setStatus( "Job Complete, maxixmum ValueCount was "
                + maxValueCount );
    }
}
```

The `reporter` field was made a class instance field, via `protected Reporter reporter`, and set in the `reduce()` method via `this.reporter = reporter`. In the `reduce()` method, the count of values is kept in `valueCount`, and if it's larger than the instance member field, `maxValueCount`, `maxValueCount` is set to it. This enables you to output the maximum number of values that shared a specific key.

In this case, the overall summary value is not particularly useful, as that value is the sum of all of the maximum values, but the per-task value is interesting and available via the web interface. A more useful solution would be to maintain an additional output file and output the key/value counts into that file.

When you select a completed or running task through the web interface (which is on port 50030 on the machine running the JobTracker, by default), you are presented with the counter summary for the job and links to detailed information about the map and reduce tasks. Each map and reduce task will have a link to the counters.

Using a Custom Partitioner

By default, the framework partitions your output based on the hash value of the key, using the `HashPartitioner` class. There are times when you need your output data partitioned differently. The standard example is a single output file where multiple output files would usually

result, which is handled by setting the number of reduce tasks to 1, via `conf.setNumReduces(1)`, or unsorted/unreduced output, which is handled via `conf.setNumReduces(0)`. If you need different partitioning, you have the option of setting a partitioner.

This chapter's example has `Long` keys. Some simple partitioner concepts could be to sort into odd/even or, if the minimum and maximum key values are known, to sort into key range-based buckets. It is also possible to partition by the value.

HOW PARTITIONING IS DONE

When the framework is performing the shuffle, each key output by the mapper is examined, and the following operation is performed:

```
int partition = partitioner.getPartition(key, value, partitions);
```

The value `partitions` is the number of reduce tasks to perform. The key, if actually output by the reducer, will end up in the output file part `partition`, with an appropriate number of leading zeros so that the file names are all the same length.

The critical issues are that the number of partitions is fixed at job start time and the partition is determined in the `output.collect()` method of the map task. The only information the partitioner has is the key, the value, the number of partitions, and whatever data was made available to it when it was instantiated.

The partitioner interface is very simple, as shown in Listing 2-7.

Listing 2-7. *The Partitioner Interface*

```
/**
 * Partitions the key space.
 *
 * <p><code>Partitioner</code> controls the partitioning of the keys of the
 * intermediate map-outputs. The key (or a subset of the key) is used to derive
 * the partition, typically by a hash function. The total number of partitions
 * is the same as the number of reduce tasks for the job. Hence this controls
 * which of the <code>m</code> reduce tasks the intermediate key (and hence the
 * record) is sent for reduction.</p>
 *
 * @see Reducer
 */
public interface Partitioner<K2, V2> extends JobConfigurable {

  /**
   * Get the paritition number for a given key (hence record) given the total
   * number of partitions i.e. number of reduce-tasks for the job.
   *
```

```
   * <p>Typically a hash function on a all or a subset of the key.</p>
   *
   * @param key the key to be paritioned.
   * @param value the entry value.
   * @param numPartitions the total number of partitions.
   * @return the partition number for the <code>key</code>.
   */
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

The `JobConfigurable` interface provides an additional `configure()` method, as the `MapReduceBase` class does.

Summary

This chapter explained what is involved in executing a MapReduce job. You now have a basic understanding of the `JobConf` object and how to use it to inform the framework of the requirements for your jobs.

You've seen how to write mapper and reducer classes, and how the `reporter` object is one of your best friends, because of the wonderful information it can provide about what is happening during the execution of your jobs. Output partitions finally make sense, and you have a sense of when and why you configure your job to reduce, and how many reducers you will use.

As a brilliant Hadoop expert, you are totally prepared to inform people of why the files they open in mapper or reducer classes are empty or short, because you know you need to close files before the framework will flush the last file system block size worth of data to disk.

In the next chapter, you'll learn how to set up of a multimachine cluster.