The Reduce phase processes the keys and their individual lists of values so that what's normally returned to the client application is a set of key/value pairs. Similar to the mapper task, which processes each record one-by-one, the reducer processes each key individually. Back in Figure 6-2, you see this concept represented as `K2,list(V2)`. The whole Reduce phase returns `list(K3,V3)`. Normally, the reducer returns a single key/value pair for every key it processes. However, these key/value pairs can be as expansive or as small as you need them to be. In the code example later in this chapter, you see a minimalist case, with a simple key/value pair with one airline code and the corresponding total number of flights completed. But in practice, you could expand the sample to return a nested set of values where, for example, you return a breakdown of the number of flights per month for every airline code.

When the reducer tasks are finished, each of them returns a results file and stores it in HDFS. As shown in Figure 6-3, the HDFS system then automatically replicates these results.

REMEMBER

Where the Resource Manager (or JobTracker if you're using Hadoop 1) tries its best to assign resources to mapper tasks to ensure that input splits are processed locally, there is no such strategy for reducer tasks. It is assumed that mapper task result sets need to be transferred over the network to be processed by the reducer tasks. This is a reasonable implementation because, with hundreds or even thousands of mapper tasks, there would be no practical way for reducer tasks to have the same locality prioritization.

Writing MapReduce Applications

The MapReduce API is written in Java, so MapReduce applications are primarily Java-based. The following list specifies the components of a MapReduce application that you can develop:

- **Driver (mandatory):** This is the application shell that's invoked from the client. It configures the MapReduce `Job` class (which you do not customize) and submits it to the Resource Manager (or JobTracker if you're using Hadoop 1).
- `Mapper` **class (mandatory):** The `Mapper` class you implement needs to define the formats of the key/value pairs you input and output as you process each record. This class has only a single method, named `map`, which is where you code how each record will be processed and what key/value to output. To output key/value pairs from the mapper task, write them to an instance of the `Context` class.

- `Reducer` **class (optional):** The reducer is optional for map-only applications where the Reduce phase isn't needed.
- `Combiner` **class (optional):** A combiner can often be defined as a reducer, but in some cases it needs to be different. (Remember, for example, that a reducer may not be able to run multiple times on a data set without mutating the results.)
- `Partitioner` **class (optional):** Customize the default partitioner to perform special tasks, such as a secondary sort on the values for each key or for rare cases involving sparse data and imbalanced output files from the mapper tasks.
- `RecordReader` **and** `RecordWriter` **classes (optional):** Hadoop has some standard data formats (for example, text files, sequence files, and databases), which are useful for many cases. For specifically formatted data, implementing your own classes for reading and writing data can greatly simplify your mapper and reducer code.

From within the driver, you can use the MapReduce API, which includes factory methods to create instances of all components in the preceding list. (In case you're not a Java person, a factory method is a tool for creating objects.)

TECHNICAL STUFF

A generic API named Hadoop Streaming lets you use other programming languages (most commonly, C, Python, and Perl). Though this API enables organizations with non-Java skills to write MapReduce code, using it has some disadvantages. Because of the additional abstraction layers that this streaming code needs to go through in order to function, there's a performance penalty and increased memory usage. Also, you can code mapper and reducer functions only with Hadoop Streaming. Record readers and writers, as well as all your partitioners, need to be written in Java. A direct consequence — and additional disadvantage — of being unable to customize record readers and writers is that Hadoop Streaming applications are well suited to handle only text-based data.

REMEMBER

In this book, we've made two critical decisions around the libraries we're using and how the applications are processed on the Hadoop cluster. We're using the MapReduce framework in the YARN processing environment (often referred to as MRv2), as opposed to the old JobTracker / TaskTracker environment from before Hadoop 2 (referred to as MRv1). Also, for the code libraries, we're using what's generally known as the new MapReduce API, which belongs to the `org.apache.hadoop.mapreduce` package. The old MapReduce API uses the `org.apache.hadoop.mapred` package. We still see code in the wild using the old API, but it's deprecated, and we don't recommend writing new applications with it.

Getting Your Feet Wet: Writing a Simple MapReduce Application

It's time to take a look at a simple application. As we do throughout this book, we'll analyze data for commercial flights in the United States. In this MapReduce application, the goal is to simply calculate the total number of flights flown for every carrier.

The FlightsByCarrier driver application

As a starting point for the FlightsByCarrier application, you need a client application driver, which is what we use to launch the MapReduce code on the Hadoop cluster. We came up with the driver application shown in Listing 6-3, which is stored in the file named `FlightsByCarrier.java`.

Listing 6-3: The FlightsByCarrier Driver Application

```
@@1
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class FlightsByCarrier {
       public static void main(String[] args) throws Exception {
              @@2
              Job job = new Job();
              job.setJarByClass(FlightsByCarrier.class);
              job.setJobName("FlightsByCarrier");

              @@3
              TextInputFormat.addInputPath(job, new Path(args[0]));
              job.setInputFormatClass(TextInputFormat.class);

              @@4
              job.setMapperClass(FlightsByCarrierMapper.class);
              job.setReducerClass(FlightsByCarrierReducer.class);

              @@5
              TextOutputFormat.setOutputPath(job, new Path(args[1]));
              job.setOutputFormatClass(TextOutputFormat.class);
              job.setOutputKeyClass(Text.class);
              job.setOutputValueClass(IntWritable.class);
```

```
            @@6
            job.waitForCompletion(true);
        }
}
```

The code in most MapReduce applications is more or less similar. The driver's job is essentially to define the structure of the MapReduce application and invoke it on the cluster — none of the application logic is defined here.

As you walk through the code, take note of these principles:

- **The import statements that follow the bold @@1 in the code pull in all required Hadoop classes.** Note that we used the new MapReduce API, as indicated by the use of the `org.apache.hadoop.mapreduce` package.
- **The first instance of the `Job` class (see the code that follows the bolded @@2) represents the entire MapReduce application.** Here, we've set the class name that will run the job and an identifier for it. By default, job properties are read from the configuration files stored in `/etc/hadoop/conf`, but you can override them by setting your `Job` class properties.
- **Using the input path we catch from the `main` method, (see the code that follows the bolded @@3), we identify the HDFS path for the data to be processed.** We also identify the expected format of the data. The default input format is `TextInputFormat` (which we've included for clarity).
- **After identifying the HDFS path, we want to define the overall structure of the MapReduce application.** We do that by specifying both the `Mapper` and `Reducer` classes. (See the code that follows the bolded @@4.) If we wanted a map-only job, we would simply omit the definition of the `Reducer` class and set the number of reduce tasks to zero with the following line of code:

  ```
  job.setNumReduceTasks(0)
  ```

- **After specifying the app's overall structure, we need to indicate the HDFS path for the application's output as well as the format of the data.** (See the code following the bolded @@5.) The data format is quite specific here because both the key and value formats need to be identified.
- **Finally, we run the job and wait.** (See the code following the bolded @@6.) The driver waits at this point until the `waitForCompletion` function returns. As an alternative, if you want your driver application to run the lines of code following the submission of the job, you can use the `submit` method instead.

The FlightsByCarrier mapper

Listing 6-4 shows the mapper code, which is stored in the file named `FlightsByCarrierMapper.java`.

Listing 6-4: The FlightsByCarrier Mapper Code

```
@@1
import java.io.IOException;
import au.com.bytecode.opencsv.CSVParser;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

@@2
public class FlightsByCarrierMapper extends
                                Mapper<LongWritable, Text, Text, IntWritable> {
       @Override
       @@3
       protected void map(LongWritable key, Text value, Context context)
                     throws IOException, InterruptedException {
              @@4
              if (key.get() > 0) {
                     String[] lines = new
              CSVParser().parseLine(value.toString());
                     @@5
                     context.write(new Text(lines[8]), new IntWritable(1));
              }
       }
}
```

The code for mappers is where you see the most variation, though it has standard boilerplate. Here are the high points:

- **The import statements that follow the bold @@1 in the code pull in all the required Hadoop classes.** The `CSVParser` class isn't a standard Hadoop class, but we use it to simply the parsing of CSV files.
- The specification of the `Mapper` class (see the code after the bolded @@2) explicitly identifies the formats of the key/value pairs that the mapper will input and output.
- The `Mapper` class has a single method, named `map`. (See the code after the bolded @@3.) The `map` method names the input key/value pair variables and the `Context` object, which is where output key/value pairs are written.

- The block of code in the if statement is where all data processing happens. (See the code after the bolded @@4.) We use the if statement to indicate that we don't want to parse the first line in the file, because it's the header information describing each column. It's also where we parse the records using the CSVParser class's parseLine method.
- With the array of strings that represent the values of the flight record being processed, the ninth value is returned to the Context object as the key. (See the code after the bolded @@5.) This value represents the carrier that completed the flight. For the value, we return a value of one because this represents one flight.

The FlightsByCarrier reducer

Listing 6-5 shows the reducer code, which is stored in the file named FlightsByCarrierReducer.java.

Listing 6-5: The FlightsByCarrier Reducer Code

```
@@1
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

@@2
public class FlightsByCarrierReducer extends
               Reducer<Text, IntWritable, Text, IntWritable> {
       @Override
       @@3
       protected void reduce(Text token, Iterable<IntWritable> counts,
                      Context context) throws IOException, InterruptedException {
               int sum = 0;

               @@4
               for (IntWritable count : counts) {
                      sum+= count.get();
               }
               @@5
               context.write(token, new IntWritable(sum));
        }
}
```

The code for reducers also has a fair amount of variation, but it also has common patterns. For example, the counting exercise is quite common. Again, here are the high points:

- The import statements that follow the bold @@1 in the code pull in all required Hadoop classes.
- The specification of the `Reducer` class (see the code after the bolded @@2) explicitly identifies the formats of the key/value pairs that the reducer will input and output.
- The `Reducer` class has a single method, named `reduce`. The `reduce` method names the input key/value pair variables and the `Context` object, which is where output key/value pairs are written. (See the code after the bolded @@3.)
- The block of code in the `for` loop is where all data processing happens. (See the code after the bolded @@4.) Remember that the `reduce` function runs on individual keys and their lists of values. So for the particular key, (in this case, the carrier), the `for` loop sums the numbers in the list, which are all ones. This provides the total number of flights for the particular carrier.
- This total is written to the context object as the value, and the input key, named `token`, is reused as the output key. (See the code after the bolded @@5.)

Running the FlightsByCarrier application

To run the FlightsByCarrier application, follow these steps:

1. **Go to the directory with your Java code and compile it using the following command:**

   ```
   javac -classpath $CLASSPATH MapRed/FlightsByCarrier/*.java
   ```

2. **Build a JAR file for the application by using this command:**

   ```
   jar cvf FlightsByCarrier.jar *.class
   ```

3. **Run the driver application by using this command:**

   ```
   hadoop jar FlightsByCarrier.jar FlightsByCarrier /user/root/airline-
             data/2008.csv /user/root/output/flightsCount
   ```

   Note that we're running the application against data from the year 2008. For this application to work, we clearly need the flight data to be stored in HDFS in the path identified in the command

```
/user/root/airline-data
```

The application runs for a few minutes. (Running it on a virtual machine on a laptop computer may take a little longer, especially if the machine has less than 8GB of RAM and only a single processor.) Listing 6-6 shows the status messages you can expect in your terminal window. You can usually safely ignore the many warnings and informational messages strewn throughout this output.

4. **Show the job's output file from HDFS by running the command**

```
hadoop fs -cat /user/root/output/flightsCount/part-r-00000
```

You see the total counts of all flights completed for each of the carriers in 2008:

```
AA        165121
AS        21406
CO        123002
DL        185813
EA        108776
HP        45399
NW        108273
PA (1)    16785
PI        116482
PS        41706
TW        69650
UA        152624
US        94814
WN        61975
```

Listing 6-6: The FlightsByCarrier Application Output

```
...
14/01/30 19:58:39 INFO mapreduce.Job: The url to track the job:
 http://localhost.localdomain:8088/proxy/application_1386752664246_0017/
14/01/30 19:58:39 INFO mapreduce.Job: Running job: job_1386752664246_0017
14/01/30 19:58:47 INFO mapreduce.Job: Job job_1386752664246_0017 running in uber
   mode : false
14/01/30 19:58:47 INFO mapreduce.Job:  map 0% reduce 0%
14/01/30 19:59:03 INFO mapreduce.Job:  map 83% reduce 0%
14/01/30 19:59:04 INFO mapreduce.Job:  map 100% reduce 0%
14/01/30 19:59:11 INFO mapreduce.Job:  map 100% reduce 100%
14/01/30 19:59:11 INFO mapreduce.Job: Job job_1386752664246_0017 completed
   successfully
14/01/30 19:59:11 INFO mapreduce.Job: Counters: 43
   File System Counters
      FILE: Number of bytes read=11873580
      FILE: Number of bytes written=23968326
      FILE: Number of read operations=0
```

(continued)

Listing 6-6 ***(continued)***

```
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=127167274
        HDFS: Number of bytes written=137
        HDFS: Number of read operations=9
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
     Job Counters
        Launched map tasks=2
        Launched reduce tasks=1
        Data-local map tasks=2
        Total time spent by all maps in occupied slots (ms)=29786
        Total time spent by all reduces in occupied slots (ms)=6024
     Map-Reduce Framework
        Map input records=1311827
        Map output records=1311826
        Map output bytes=9249922
        Map output materialized bytes=11873586
        Input split bytes=236
        Combine input records=0
        Combine output records=0
        Reduce input groups=14
        Reduce shuffle bytes=11873586
        Reduce input records=1311826
        Reduce output records=14
        Spilled Records=2623652
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=222
        CPU time spent (ms)=8700
        Physical memory (bytes) snapshot=641634304
        Virtual memory (bytes) snapshot=2531708928
        Total committed heap usage (bytes)=496631808
     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=127167038
     File Output Format Counters
 Bytes Written=137
```

There you have it. You've just seen how to program and run a basic MapReduce application. What we've done is read the flight data set and calculated the total number of flights flown for every carrier. To make this work in MapReduce, we had to think about how to program this calculation so that the individual pieces of the larger data set could be processed in parallel. And, not to put too fine a point on it, the thoughts we came up with turned out to be pretty darn good!