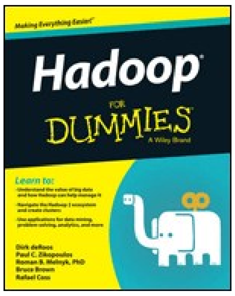

Hadoop for Dummies

by Dirk deRoos et al.

Chapter 6: MapReduce Programming

In This Chapter

- Thinking in parallel
- Working with key/value pairs
- Tracking your application flow
- Running the sample MapReduce application

After you've stored reams and reams of data in HDFS (a distributed storage system spread over an expandable cluster of individual slave nodes), the first question that comes to mind is "How can I analyze or query my data?" Transferring all this data to a central node for processing isn't the answer, since you'll be waiting forever for the data to transfer over the network (not to mention waiting for everything to be processed serially). So what's the solution? MapReduce!

As we describe in Chapter 1, Google faced this exact problem with their distributed Google File System (GFS), and came up with their MapReduce data processing model as the best possible solution. Google needed to be able to grow their data storage and processing capacity, and the only feasible model was a distributed system. In Chapter 4, we look at a number of the benefits of storing data in the Hadoop Distributed File System (HDFS): low cost, fault-tolerant, and easily scalable, to name just a few. In Hadoop, MapReduce integrates with HDFS to provide the exact same benefits for data processing.

At first glance, the strengths of Hadoop sound too good to be true — and overall the strengths truly are good! But there is a cost here: writing applications for distributed systems is completely different from writing the same code for centralized systems. For applications to take advantage of the distributed slave nodes in the Hadoop cluster, the application logic will need to run in parallel.

Thinking in Parallel

Let's say you want to do something simple, like count the number of flights for each carrier in our flight data set — this will be our example scenario for this chapter. For a normal program that runs serially, this is a simple operation. Listing 6-1 shows the pseudocode, which is fairly straightforward: set up the array to store the number of times you run across each carrier, and then, as you read each record in sequence, increment the applicable airline's counter.

Listing 6-1: Pseudocode for Calculating The Number of Flights By Carrier Serially

```
create a two-dimensional array
  create a row for every airline carrier
    populate the first column with the carrier code
    populate the second column with the integer zero

for each line of flight data
  read the airline carrier code
  find the row in the array that matches the carrier code
    increment the counter in the second column by one

print the totals for each row in the two-dimensional array
```

The thing is, you would not be able to take this (elegantly simple) code and run it successfully on flight data stored in a distributed system. Even though this is a simple example, you need to think in parallel as you code your application. Listing 6-2 shows the pseudocode for calculating the number of flights by carrier in parallel.

Listing 6-2: Pesudocode for Calculating The Number of Flights By Carrier in Parallel

```
Map Phase:
  for each line of flight data
    read the current record and extract the airline carrier code
    output the airline carrier code and the number one as a key/value pair

Shuffle and Sort Phase:
  read the list of key/value pairs from the map phase
  group all the values for each key together
    each key has a corresponding array of values
  sort the data by key
  output each key and its array of values

Reduce Phase:
  read the list of carriers and arrays of values from the shuffle and sort phase
  for each carrier code
    add the total number of ones in the carrier code's array of values together

print the totals for each row in the two-dimensional array
```

The code in Listing 6-2 shows a completely different way of thinking about how to process data. Since we need totals, we had to break this application up into phases. The first phase is the *map phase*, which is where every record in the data set is processed individually. Here, we extract the carrier code from the flight data record it's assigned, and then export a key/value pair, with the carrier code as the key and the value being an integer `one`. The map operation is run against every record in the data set. After every record is processed, you need to ensure that all the values (the integer `ones`) are grouped together for each key, which is the airline carrier code, and then sorted by key. This is known as the *shuffle and sort phase*. Finally, there is the reduce phase, where you add the total number of ones together for each airline carrier, which gives you the total flights for each airline carrier.

As you can see, there is little in common between the serial version of the code and the parallel version. Also, even though this is a simple example, developing the parallel version requires an altogether different approach. What's more, as the computation problems get even a little more difficult, they become even harder when they need to be parallelized.

Seeing the Importance of MapReduce

For most of Hadoop's history, MapReduce has been the only game in town when it comes to data processing. The availability of MapReduce has been the reason for Hadoop's success and at the same time a major factor in limiting further adoption.

As we'll see later in this chapter, MapReduce enables skilled programmers to write distributed applications without having to worry about the underlying distributed computing infrastructure. This is a very big deal: Hadoop and the MapReduce framework handle all sorts of complexity that application developers don't need to handle. For example, the ability to transparently scale out the cluster by adding nodes and the automatic failover of both data storage and data processing subsystems happen with zero impact on applications.

The other side of the coin here is that although MapReduce hides a tremendous amount of complexity, you can't afford to forget what it is: an interface for parallel programming. This is an advanced skill — and a barrier to wider adoption. There simply aren't yet many MapReduce programmers, and not everyone has the skill to master it.

The goal of this chapter is to help you understand how MapReduce applications work, how to think in parallel, and to provide a basic entry point into the world of MapReduce programming.

REMEMBER In Hadoop's early days (Hadoop 1 and before), you could only run MapReduce applications on your clusters. In Hadoop 2, the YARN component changed all that by taking over resource management and scheduling from the MapReduce framework, and providing a generic interface to facilitate applications to run on a Hadoop cluster. (See Chapter 7 for our discussion of YARN's framework-agnostic resource management.) In short, this means MapReduce is now just one of many application frameworks you can use to develop and run applications on Hadoop. Though it's certainly possible to run applications using other frameworks on Hadoop, it doesn't mean that we can start forgetting about MapReduce. At the time we wrote this book, MapReduce was still the only production-ready data processing framework available for Hadoop. Though other frameworks will eventually become available, MapReduce has almost a decade of maturity under its belt (with almost 4,000 JIRA issues completed, involving hundreds of developers, if you're keeping track). There's no dispute: MapReduce is Hadoop's most mature framework for data processing. In addition, a significant amount of MapReduce code is now in use that's unlikely to go anywhere soon. Long story short: MapReduce is an important part of the Hadoop story.

Later in this book, we cover certain programming abstractions to MapReduce, such as Pig (see Chapter 8) and Hive (see Chapter 13), which hide the complexity of parallel programming. The Apache Hive and Apache Pig projects are highly popular because they're easier entry points for data processing on Hadoop. For many problems, especially the kinds that you can solve with SQL, Hive and Pig are excellent tools. But for a wider-ranging task such as statistical processing or text extraction, and especially for processing unstructured data, you need to use MapReduce.

Doing Things in Parallel: Breaking Big Problems into Many Bite-Size Pieces

If you're a programmer, chances are good that you're at least aware of reddit, a popular discussion site — perhaps you're even a full-blown redditor. Its Ask Me Anything subreddit features a notable person logging in to reddit to answer redditor's questions. In a running gag, someone inevitably asks the question, "Would you rather fight 1 horse-sized duck or 100 duck-sized horses?" The answers and the rationale behind them are sources of great amusement, but they create a mental picture of what Hadoop and MapReduce are all about: scaling out as opposed to scaling up. Of course you'd rather defend yourself against 1 horse-sized duck — a herd of duck-sized horses would overwhelm you in seconds!

Looking at MapReduce Application Flow

At its core, MapReduce is a programming model for processing data sets that are stored in a distributed manner across a Hadoop cluster's slave nodes. The key concept here is *divide and conquer*. Specifically, you want to break a large data set into many smaller pieces and process them in parallel with the same algorithm. With the Hadoop Distributed File System (HDFS), the files are already divided into bite-sized pieces. MapReduce is what you use to process all the pieces.

MapReduce applications have multiple phases, as spelled out in this list:

1. Determine the exact data sets to process from the data blocks. This involves calculating where the records to be processed are located

within the data blocks.

2. Run the specified algorithm against each record in the data set until all the records are processed. The individual instance of the application running against a block of data in a data set is known as a *mapper task*. (This is the mapping part of MapReduce.)
3. Locally perform an interim reduction of the output of each mapper. (The outputs are provisionally combined, in other words.) This phase is optional because, in some common cases, it isn't desirable.
4. Based on partitioning requirements, group the applicable partitions of data from each mapper's result sets.
5. Boil down the result sets from the mappers into a single result set — the Reduce part of MapReduce. An individual instance of the application running against mapper output data is known as a *reducer task*. (As strange as it may seem, since "Reduce" is part of the MapReduce name, this phase can be optional; applications without a reducer are known as *map-only jobs*, which can be useful when there's no need to combine the result sets from the map tasks.)

Understanding Input Splits

The way HDFS has been set up, it breaks down very large files into large blocks (for example, measuring 128MB), and stores three copies of these blocks on different nodes in the cluster. HDFS has no awareness of the content of these files. (If this business about HDFS doesn't ring a bell, check out Chapter 4.)

In YARN, when a MapReduce job is started, the Resource Manager (the cluster resource management and job scheduling facility) creates an Application Master daemon to look after the lifecycle of the job. (In Hadoop 1, the JobTracker monitored individual jobs as well as handling job scheduling and cluster resource management. For more on this, see Chapter 7.) One of the first things the Application Master does is determine which file blocks are needed for processing. The Application Master requests details from the NameNode on where the replicas of the needed data blocks are stored. Using the location data for the file blocks, the Application Master makes requests to the Resource Manager to have map tasks process specific blocks on the slave nodes where they're stored.

REMEMBER The key to efficient MapReduce processing is that, wherever possible, data is processed *locally* — on the slave node where it's stored.

Before looking at how the data blocks are processed, you need to look more closely at how Hadoop stores data. In Hadoop, files are composed of individual records, which are ultimately processed one-by-one by mapper tasks. For example, the sample data set we use in this book contains information about completed flights within the United States between 1987 and 2008. We have one large file for each year, and within every file, each individual line represents a single flight. In other words, one line represents one record. Now, remember that the block size for the Hadoop cluster is 64MB, which means that the light data files are broken into chunks of exactly 64MB.

Do you see the problem? If each map task processes all records in a specific data block, what happens to those records that span block boundaries? File blocks are exactly 64MB (or whatever you set the block size to be), and because HDFS has no conception of what's inside the file blocks, it can't gauge when a record might spill over into another block. To solve this problem, Hadoop uses a logical representation of the data stored in file blocks, known as *input splits*. When a MapReduce job client calculates the input splits, it figures out where the first whole record in a block begins and where the last record in the block ends. In cases where the last record in a block is incomplete, the input split includes location information for the next block and the byte offset of the data needed to complete the record. Figure 6-1 shows this relationship between data blocks and input splits.

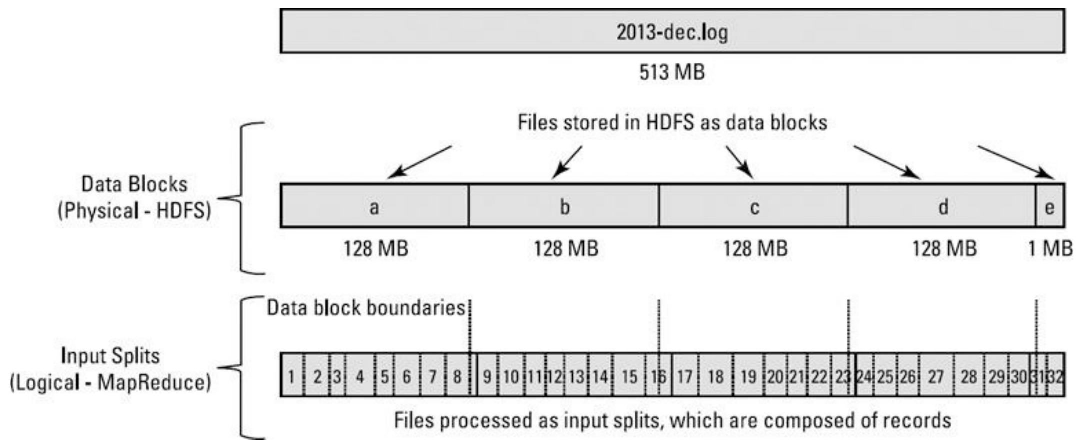
2013-dec.log
513 MB
Files stored in HDFS as data blocks
Data Blocks (Physical - HDFS)
a
b
c
d
e
128 MB
128 MB
128 MB
128 MB
1 MB
Data block boundaries
Input Splits (Logical - MapReduce)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Files processed as input splits, which are composed of records

Figure 6-1: Data blocks (HDFS) and input splits (MapReduce)

TIP You can configure the Application Master daemon (or JobTracker, if you're in Hadoop 1) to calculate the input splits instead of the job client, which would be faster for jobs processing a large number of data blocks.

MapReduce data processing is driven by this concept of input splits. The number of input splits that are calculated for a specific application determines the number of mapper tasks. Each of these mapper tasks is assigned, where possible, to a slave node where the input split is stored. The Resource Manager (or JobTracker, if you're in Hadoop 1) does its best to ensure that input splits are processed locally.

Seeing How Key/Value Pairs Fit into the MapReduce Application Flow

You may be wondering what happens in the processing of all these input splits. To answer this question, you need to understand that a MapReduce application processes the data in input splits on a *record-by-record* basis and that each record is understood by MapReduce to be a *key/value* pair. (In more technical descriptions of Hadoop, you see key/value pairs referred to as *tuples*.)

TIP Obviously, when you're processing data, not everything needs to be represented as a key/value pair, so in cases where it isn't needed, you can provide a dummy key or value.

We describe the phases of a MapReduce application in the "Looking at MapReduce application flow" section, earlier in this chapter. Figure 6-2 fills out that description by showing how our sample MapReduce application (complete with sample flight data) makes its way through these phases. The next few sections of this chapter walk you through the process shown in Figure 6-2.

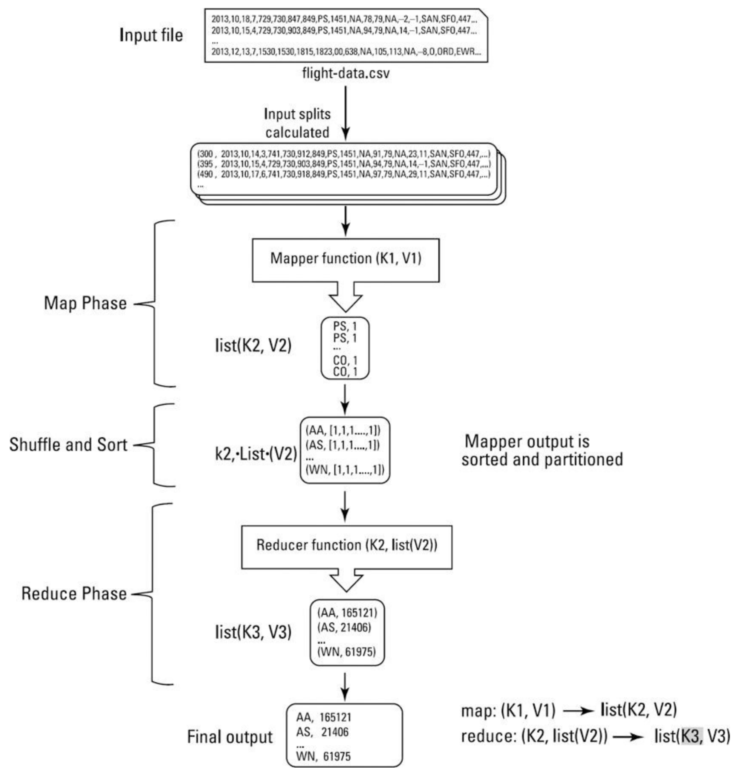
Input file
2013,10,18,7,729,730,847,849,PS,1451,NA,78,79,NA,–2,–1,SAN,SFO,447...
2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447...
...
2013,12,13,7,1530,1530,1815,1823,00,638,NA,105,113,NA,–8,0,ORD,EWR...
flight-data.csv
Input splits calculated
(300 , 2013,10,14,3,741,730,912,849,PS,1451,NA,91,79,NA,23,11,SAN,SFO,447,...)
(395 , 2013,10,15,4,729,730,903,849,PS,1451,NA,94,79,NA,14,–1,SAN,SFO,447,...)
(490 , 2013,10,17,6,741,730,918,849,PS,1451,NA,97,79,NA,29,11,SAN,SFO,447,...)
...
Map Phase
Mapper function (K1, V1)
list(K2, V2)
PS, 1
PS, 1
...
CO, 1
CO, 1
Shuffle and Sort
k2,·List·(V2)
(AA, [1,1,1....,1])
(AS, [1,1,1....,1])
...
(WN, [1,1,1....,1])
Mapper output is sorted and partitioned
Reduce Phase
Reducer function (K2, list(V2))
list(K3, V3)
(AA, 165121)
(AS, 21406)
...
(WN, 61975)
Final output
AA, 165121
AS, 21406
...
WN, 61975
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

Figure 6-2: Data flow through the MapReduce cycle

Map Phase

After the input splits have been calculated, the mapper tasks can start processing them — that is, right after the Resource Manager's scheduling facility assigns them their processing resources. (In Hadoop 1, the JobTracker assigns mapper tasks to specific processing slots.) The mapper task itself processes its input split one record at a time — in Figure 6-2, this lone record is represented by the key/value pair `(K1,V1)`. In the case of our flight data, when the input splits are calculated (using the default file processing method for text files), the assumption is that each row in the text file is a single record. For each record, the text of the row itself represents the value, and the byte offset of each row from the beginning of the split is considered to be the key.

TECHNICAL STUFF You might be wondering why the row number isn't used instead of the byte offset. When you consider that a very large text file is broken down into many individual data blocks, and is processed as many splits, the row number is a risky concept. The number of lines in each split vary, so it would be impossible to compute the number of rows preceding the one being processed. However, with the byte offset, you can be precise, because every block has a fixed number of bytes.

As a mapper task processes each record, it generates a new key/value pair: `(K2,V2)`. The key and the value here can be completely different from the input pair. The output of the mapper task is the full collection of all these key/value pairs. In Figure 6-2, the output is represented by `list(K2,V2)`. Before the final output file for each mapper task is written, the output is partitioned based on the key and sorted. This partitioning means that all of the values for each key are grouped together, resulting in the following output: `K2,list(V2)`.

In the case of our fairly basic sample application, there is only a single reducer, so all the output of the mapper task is written to a single file. But in cases with multiple reducers, every mapper task may generate multiple output files as well. The breakdown of these output files is based on the partitioning key. For example, if there are only three distinct partitioning keys output for the mapper tasks and you have configured three reducers for the job, there will be three mapper output files. In this example, if a particular mapper task processes an input split and it generates output with two of the three keys, there will be only two output files.

TIP Always compress your mapper tasks' output files. The biggest benefit here is in performance gains, because writing smaller output files minimizes the inevitable cost of transferring the mapper output to the nodes where the reducers are running. Enable compression by setting the `mapreduce.map.output.compress` property to `true` and assigning a compression codec to the `mapred.map.output.compress.codec` property. (This property can be found in the `mapred-site.xml` file, which is stored in Hadoop's `conf` directory. For details on configuring Hadoop, see Chapter 3.)

TECHNICAL STUFF The default partitioner is more than adequate in most situations, but sometimes you may want to customize how the data is partitioned before it's processed by the reducers. For example, you may want the data in your result sets to be sorted by the key and their values — known as a *secondary* sort. To do this, you can override the default partitioner and implement your own. This process requires some care, however, because you'll want to ensure that the number of records in each partition is uniform. (If one reducer has to process much more data than the other reducers, you'll wait for your MapReduce job to finish while the single overworked reducer is slogging through its disproportionally large data set.) Using uniformly sized intermediate files, you can better take advantage of the parallelism available in MapReduce processing.

Shuffle Phase

After the Map phase and before the beginning of the Reduce phase is a handoff process, known as *shuffle and sort*. Here, data from the mapper tasks is prepared and moved to the nodes where the reducer tasks will be run. When the mapper task is complete, the results are sorted by key, partitioned if there are multiple reducers, and then written to disk. You can see this concept in Figure 6-3, which shows the MapReduce data processing flow and its interaction with the physical components of the Hadoop cluster. (One quick note about Figure 6-3: Data in memory is represented by white squares, and data stored to disk is represented by gray squares.) To speed up the overall MapReduce process, data is immediately moved to the reducer tasks' nodes, to avoid a flood of network activity when the final mapper task finishes its work. This transfer happens while the mapper task is running, as the outputs for each record — remember `(K2,V2)` — are stored in the memory of a waiting reducer task. (You can configure whether this happens — or doesn't happen — and also the number of threads involved.) Keep in mind that even though a reducer task might have most of the mapper task's output, the reduce task's processing cannot begin until all mapper tasks have finished.

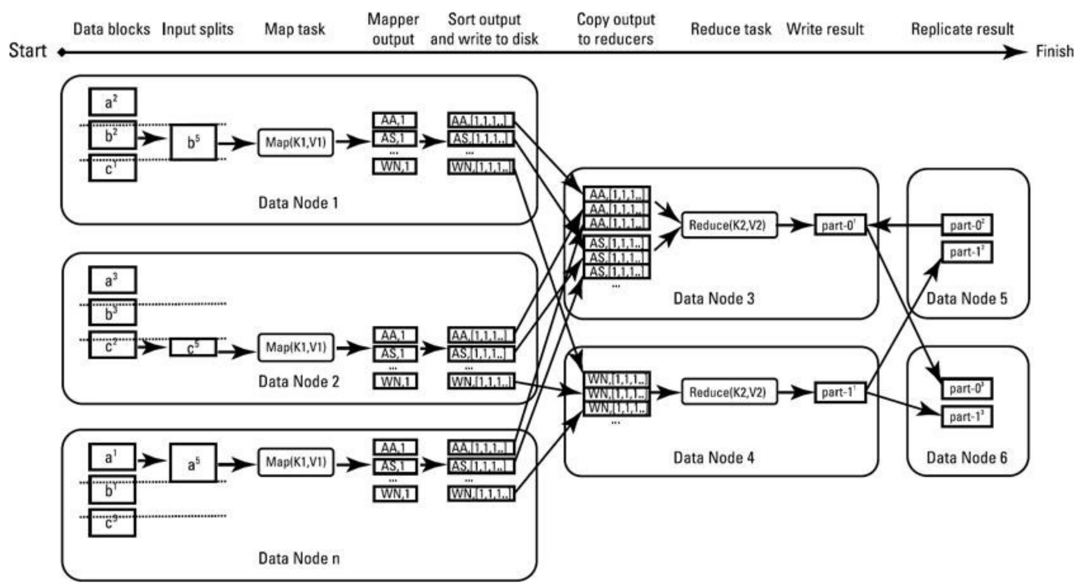

Figure 6-3: MapReduce processing flow

TECHNICAL STUFF To avoid scenarios where the performance of a MapReduce job is hampered by one straggling mapper task that's running on a poorly performing slave node, the MapReduce framework uses a concept called *speculative execution*. In case some mapper tasks are running slower than what's considered reasonable, the Application Master will spawn duplicate tasks (in Hadoop 1, the JobTracker does this). Whichever task finishes first — the duplicate or the original — its results are stored to disk, and the other task is killed. If you're monitoring your jobs closely and are wondering why there are more mapper tasks running than you expect, this is a likely reason.

The output from mapper tasks isn't written to HDFS, but rather to local disk on the slave node where the mapper task was run. As such, it's not replicated across the Hadoop cluster.

TIP Aside from compressing the output, you can potentially boost performance by running a combiner task. This simple tactic, shown in Figure 6-4, involves performing a local reduce of the output for individual mapper tasks. In the majority of cases, no extra programming is needed, as you can tell the system to use the reducer function. If you're not using your reducer function, you need to ensure that the combiner function's output is identical to that of the reducer function. It's up to the MapReduce framework whether the combiner function needs to be run once, multiple times, or never, so it's critical that the combiner's code ensures that the final results are unaffected by multiple runs. Running the combiner can yield a performance benefit by lessening the amount of intermediate data that would otherwise need to be transferred over the network. This also lowers the amount of processing the reducer tasks would need to do. You are running an extra task here, so it is possible that any performance gain is negligible or may even result in worse overall performance. Your mileage may vary, so we recommend testing this carefully.

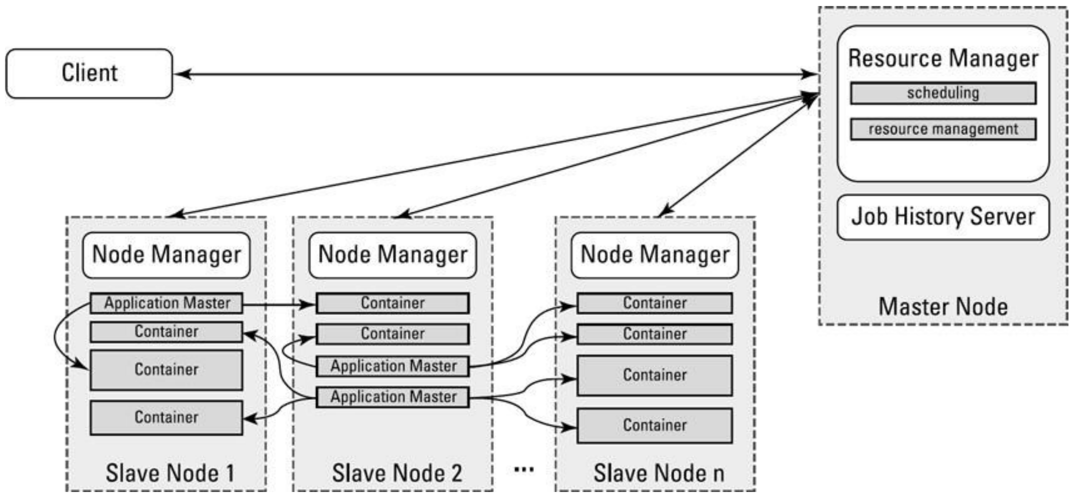

Figure 6-4: Reducing intermediate data size with combiners

After all the results of the mapper tasks are copied to the reducer tasks' nodes, these files are merged and sorted.

Reduce Phase

Here's the blow-by-blow so far: A large data set has been broken down into smaller pieces, called *input splits*, and individual instances of mapper tasks have processed each one of them. In some cases, this single phase of processing is all that's needed to generate the desired application output. For example, if you're running a basic transformation operation on the data — converting all text to uppercase, for example, or extracting key frames from video files — the lone phase is all you need. (This is known as a *map-only* job, by the way.) But in many other cases, the job is only half-done when the mapper tasks have written their output. The remaining task is boiling down all interim results to a single, unified answer.

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
       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!