UNIT-II
Map Reduce

- A Weather Dataset
  - Data Formate
- Analyzing the Data with Unix Tools
- Analyzing the Data with Hadoop
  - Map and Reduce
  - Java MapReduce
- Scaling Out
  - Data flow
  - Combiner Functions
  - Running a Distributed Mapreduce Job

MapReduce is a programming model for data processing. Hadoop can run MapReduce programs written in various languages; MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hands of anyone with enough machines at their disposal. MapReduce comes into its own for large datasets.

A Weather Dataset

Weather sensors collecting data every hour at many locations across the globe gather a large volume of log data, which is a good candidate for analysis with MapReduce, since it is semistructured and record-oriented.

The data used is from the National Climatic Data Center (NCDC, *http://www.ncdc.noaa.gov/*). The data is stored using a line-oriented ASCII format, in which each line is a record. The format supports a rich set of meteorological elements, many of which are optional or with variable data lengths. For simplicity, focus on the basic elements, such as temperature, which are always present and are of fixed width.

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

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

0057

332130 # USAF weather station identifier
99999 # WBAN weather station identifier
19500101 # observation date
0300 # observation time
4
+51317 # latitude (degrees x 1000)
+028783 # longitude (degrees x 1000)
FM-12
+0171 # elevation (meters)
99999
V020
320 # wind direction (degrees)
1 # quality code
N
0072
1
00450 # sky ceiling height (meters)
1 # quality code
CN
010000 # visibility distance (meters)
1 # quality code
N9
-0128 # air temperature (degrees Celsius x 10)
1 # quality code
-0139 # dew point temperature (degrees Celsius x 10)
1 # quality code
10268 # atmospheric pressure (hectopascals x 10)
1 # quality code

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

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

Since there are tens of thousands of weather stations, the whole dataset is made up of a large number of relatively small files. It's generally easier and more efficient to process a smaller number of relatively large files, so the data was preprocessed so that each year's readings were concatenated into a single file.

Analyzing the Data with Unix Tools

The classic tool for processing line-oriented data is *awk*. Example 2-2 is a small script to calculate the maximum temperature for each year.

Example 2-2. A program for finding the maximum recorded temperature by year from NCDC weather records

```
#!/usr/bin/env bash
for year in all/*
do
echo -ne `basename $year .gz`"\t"
gunzip -c $year | \
awk '{ temp = substr($0, 88, 5) + 0;
q = substr($0, 93, 1);
if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
END { print max }'
Done
```

The script loops through the compressed year files, first printing the year, and then processing each file using *awk*. The *awk* script extracts two fields from the data: the air temperature and the quality code. The air temperature value s turned into an integer by adding 0. Next, a test is applied to see if the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and if the quality code indicates that the reading is not suspect or erroneous. If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found. The END block is executed after all the lines in the file have been processed, and it prints the maximum value.

Here is the beginning of a run:

```
% ./max_temperature.sh
1901 317
1902 244
1903 289
1904 256
1905 283
...
```

The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901 (there were very few readings at the beginning of the century, so this is plausible). The complete run for the century took 42 minutes in one run on a single EC2 High-CPU Extra Large Instance.

To speed up the processing, need to run parts of the program in parallel. Process different years in different processes, using all the available hardware threads on a machine. There are a few problems with this,
however.

First, dividing the work into equal-size pieces isn't always easy or obvious. the file size for different years varies widely, so some processes will finish much earlier than others. Even if they pick up further work, the whole run is dominated by the longest file. A better approach, although one that requires more work, is to split the input into fixed-size chunks and assign each chunk to a process.

Second, combining the results from independent processes may need further processing. In this case, the result for each year is independent of other years and may be combined by concatenating all the results, and sorting by year. If using the fixed-size chunk approach, the combination is more delicate.

Third, still limited by the processing capacity of a single machine. If the best time to achieve is 20 minutes with the number of processors to have, then that's it. You can't make it go faster. Also, some datasets grow beyond the capacity of a single machine. When we start using multiple machines, a whole host of other factors come into play, mainly falling in the category of coordination and reliability. Who runs the overall job? How do we deal with failed processes?So, though it's feasible to parallelize the processing, in practice it's messy. Using a framework like Hadoop to take care of these issues.

Analyzing the Data with Hadoop

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

Map and Reduce

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function. The input to our map phase is the raw NCDC data. We choose a text input format that gives us each line in the dataset as a text value. The key is the offset of the beginning of the line from the beginning of the file.

Our map function is simple. Pull out the year and the air temperature, since these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reducer function can do its work on it: finding the maximum temperature for each year. The map

function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.
To visualize the way the map works, consider the following sample lines of input data
(some unused columns have been dropped to fit the page, indicated by ellipses):

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

These lines are presented to the map function as the key-value pairs:

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The keys are the line offsets within the file, which we ignore in our map function. The map function merely extracts the year and the air temperature (indicated in bold text), and emits them as its output (the temperature values have been interpreted as integers):
(1950, 0)
(1950, 22)
(1950, −11)
(1949, 111)
(1949, 78)
The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:
(1949, [111, 78])
(1950, [0, 22, −11])

Each year appears with a list of all its air temperature readings. All the reduce function has to do now is iterate through the list and pick up the maximum reading:
(1949, 111)
(1950, 22)

This is the final output: the maximum global temperature recorded in each year. The whole data flow is illustrated in Figure 2-1. At the bottom of the diagram is a Unix pipeline, which mimics the whole MapReduce flow, and which we will see again later in the chapter when we look at Hadoop Streaming.

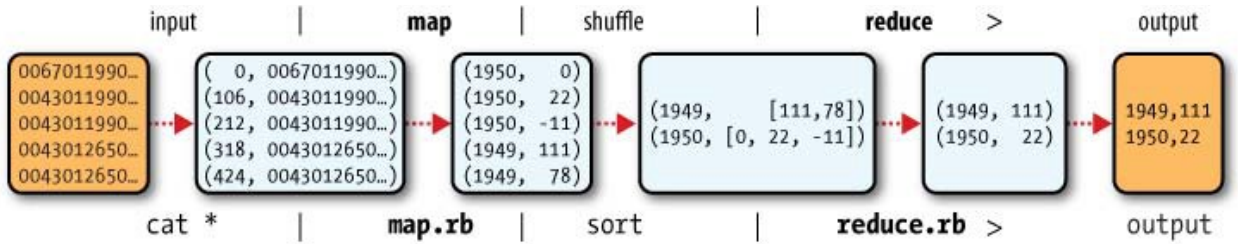

Figure 2-1. MapReduce logical data flow

Java MapReduce

Having run through how the MapReduce program works, the next step is to express it in code.
We need three things:
a map function,
a reduce function, and
some code to run the job.

The map function is represented by an implementation of the Mapper interface, which declares a map() method. Example 2-3 shows the implementation of our map function.
Example 2-3. Mapper for maximum temperature example

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class MaxTemperatureMapper extends MapReduceBase
implements Mapper<LongWritable, Text, Text, IntWritable> {
private static final int MISSING = 9999;

public void map(LongWritable key, Text value,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
String line = value.toString();
String year = line.substring(15, 19);
int airTemperature;
if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
airTemperature = Integer.parseInt(line.substring(88, 92));
} else {
airTemperature = Integer.parseInt(line.substring(87, 92));
```

```
}
String quality = line.substring(92, 93);
if (airTemperature != MISSING && quality.matches("[01459]")) {
output.collect(new Text(year), new IntWritable(airTemperature));
}
}
}
```

The Mapper interface is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function.
For the present example, the input key is a long integer offset, the input value is a line of text, the output key is a year, and the output value is an air temperature (an integer).

Rather than use built-in Java types, Hadoop provides its own set of basic types that are optimized for network serialization. These are found in the org.apache.hadoop.io package.
Here we use LongWritable, which corresponds to a Java Long, Text (like Java String), and IntWritable (like Java Integer).
The map() method is passed a key and a value. We convert the Text value containing the line of input into a Java String, then use its substring() method to extract the columns we are interested in.
The map() method also provides an instance of OutputCollector to write the output to.
In this case, we write the year as a Text object (since we are just using it as a key), and the temperature is wrapped in an IntWritable. We write an output record only if the temperature is present and the quality code indicates the temperature reading is OK.
The reduce function is similarly defined using a Reducer, as illustrated in Example 2-4.

Example 2-4. Reducer for maximum temperature example

```
import java.io.IOE1xception;
import java.util.Iterator;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
public class MaxTemperatureReducer extends MapReduceBase
implements Reducer<Text, IntWritable, Text, IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException {
int maxValue = Integer.MIN_VALUE;
while (values.hasNext()) {
maxValue = Math.max(maxValue, values.next().get());
}
output.collect(key, new IntWritable(maxValue));
}
}
```

Example 2-5. Application to find the maximum temperature in the weather dataset

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
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;
public class MaxTemperature {
public static void main(String[] args) throws IOException {
if (args.length != 2) {
System.err.println("Usage: MaxTemperature <input path> <output path>");
System.exit(-1);
}
JobConf conf = new JobConf(MaxTemperature.class);
conf.setJobName("Max temperature");
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
conf.setMapperClass(MaxTemperatureMapper.class);
conf.setReducerClass(MaxTemperatureReducer.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
JobClient.runJob(conf);
}
```

}
A JobConf object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass a class in the JobConf constructor, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.
Having constructed a JobConf object, we specify the input and output paths. An input path is specified by calling the static addInputPath() method on FileInputFormat, and it can be a single file, a directory (in which case, the input forms all the files in that directory), or a file pattern. As the name suggests, addInputPath() can be called more than once to use input from multiple paths.

The output path (of which there is only one) is specified by the static setOutput Path() method on FileOutputFormat. It specifies a directory where the output files from the reducer functions are written. The directory shouldn't exist before running the job, as Hadoop will complain and not run the job. This precaution is to prevent data loss (it can be very annoying to accidentally overwrite the output of a long job with another).
Next, we specify the map and reduce types to use via the setMapperClass() and setReducerClass() methods.
The setOutputKeyClass() and setOutputValueClass() methods control the output types for the map and the reduce functions, which are often the same, as they are in our case. If they are different, then the map output types can be set using the methods setMapOutputKeyClass() and setMapOutputValueClass(). The input types are controlled via the input format, which we have not explicitly set since we are using the default TextInputFormat. After setting the classes that define the map and reduce functions, we are ready to run the job. The static runJob() method on JobClient submits the job and waits for it to finish, writing information about its progress to the console.

A test run
After writing a MapReduce job, it's normal to try it out on a small dataset to flush out any immediate problems with the code. First install Hadoop in standalone mode— there are instructions for how to do this in Appendix A. This is the mode in which Hadoop runs using the local filesystem with a local job runner. Let's test it on the fiveline sample discussed earlier (the output has been slightly reformatted to fit the page):
% **export HADOOP_CLASSPATH=build/classes**
% **hadoop MaxTemperature input/ncdc/sample.txt output**
09/04/07 12:34:35 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=Job
Tracker, sessionId=
09/04/07 12:34:35 WARN mapred.JobClient: Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.

09/04/07 12:34:35 WARN mapred.JobClient: No job jar file set. User classes may not
be found. See JobConf(Class) or JobConf#setJar(String).
09/04/07 12:34:35 INFO mapred.FileInputFormat: Total input paths to process : 1
09/04/07 12:34:35 INFO mapred.JobClient: Running job: job_local_0001
09/04/07 12:34:35 INFO mapred.FileInputFormat: Total input paths to process : 1
09/04/07 12:34:35 INFO mapred.MapTask: numReduceTasks: 1
09/04/07 12:34:35 INFO mapred.MapTask: io.sort.mb = 100
09/04/07 12:34:35 INFO mapred.MapTask: data buffer = 79691776/99614720
09/04/07 12:34:35 INFO mapred.MapTask: record buffer = 262144/327680
09/04/07 12:34:35 INFO mapred.MapTask: Starting flush of map output
09/04/07 12:34:36 INFO mapred.MapTask: Finished spill 0
09/04/07 12:34:36 INFO mapred.TaskRunner: Task:attempt_local_0001_m_000000_0 is
done. And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner: file:/Users/tom/workspace/htdg/input/n
cdc/sample.txt:0+529
09/04/07 12:34:36 INFO mapred.TaskRunner: Task 'attempt_local_0001_m_000000_0' done.
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.Merger: Merging 1 sorted segments
09/04/07 12:34:36 INFO mapred.Merger: Down to the last merge-pass, with 1 segments
left of total size: 57 bytes
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner: Task:attempt_local_0001_r_000000_0 is done.
And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner: Task attempt_local_0001_r_000000_0 is
allowed to commit now
09/04/07 12:34:36 INFO mapred.FileOutputCommitter: Saved output of task
'attempt_local_0001_r_000000_0' to file:/Users/tom/workspace/htdg/output
09/04/07 12:34:36 INFO mapred.LocalJobRunner: reduce > reduce
09/04/07 12:34:36 INFO mapred.TaskRunner: Task 'attempt_local_0001_r_000000_0' done.
09/04/07 12:34:36 INFO mapred.JobClient: map 100% reduce 100%
09/04/07 12:34:36 INFO mapred.JobClient: Job complete: job_local_0001
09/04/07 12:34:36 INFO mapred.JobClient: Counters: 13
09/04/07 12:34:36 INFO mapred.JobClient: FileSystemCounters
09/04/07 12:34:36 INFO mapred.JobClient: FILE_BYTES_READ=27571
09/04/07 12:34:36 INFO mapred.JobClient: FILE_BYTES_WRITTEN=53907
09/04/07 12:34:36 INFO mapred.JobClient: Map-Reduce Framework
09/04/07 12:34:36 INFO mapred.JobClient: Reduce input groups=2

09/04/07 12:34:36 INFO mapred.JobClient: Combine output records=0
09/04/07 12:34:36 INFO mapred.JobClient: Map input records=5
09/04/07 12:34:36 INFO mapred.JobClient: Reduce shuffle bytes=0
09/04/07 12:34:36 INFO mapred.JobClient: Reduce output records=2
09/04/07 12:34:36 INFO mapred.JobClient: Spilled Records=10
09/04/07 12:34:36 INFO mapred.JobClient: Map output bytes=45
09/04/07 12:34:36 INFO mapred.JobClient: Map input bytes=529
09/04/07 12:34:36 INFO mapred.JobClient: Combine input records=0
09/04/07 12:34:36 INFO mapred.JobClient: Map output records=5
09/04/07 12:34:36 INFO mapred.JobClient: Reduce input records=5

When the hadoop command is invoked with a classname as the first argument, it launches a JVM to run the class. It is more convenient to use hadoop than straight java since the former adds the Hadoop libraries (and their dependencies) to the classpath and picks up the Hadoop configuration, too. To add the application classes to the classpath, we've defined an environment variable called HADOOP_CLASSPATH, which the hadoop script picks up.

Scaling Out

- Data flow
- Combiner Functions
- Running a Distributed Mapreduce Job

Data Flow

A MapReduce *job* is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into *tasks*, of which there are two types:

***map tasks* and *reduce tasks*.**

There are two types of nodes that control the job execution process: a *jobtracker* and a number of *tasktrackers*.

The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers.

Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job.

If a task fails, the jobtracker can reschedule it on a different tasktracker. Hadoop divides the input to a MapReduce job into fixed-size pieces called *input splits*, or just *splits*. Hadoop creates one map task for each split, which runs the userdefined map function for each *record* in the split.

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

On the other hand, if splits are too small, then the overhead of managing the splits and of map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created.

Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the *data locality optimization*. It should now

be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.
Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes. The whole data flow with a single reduce task is illustrated in Figure 2-2. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes.

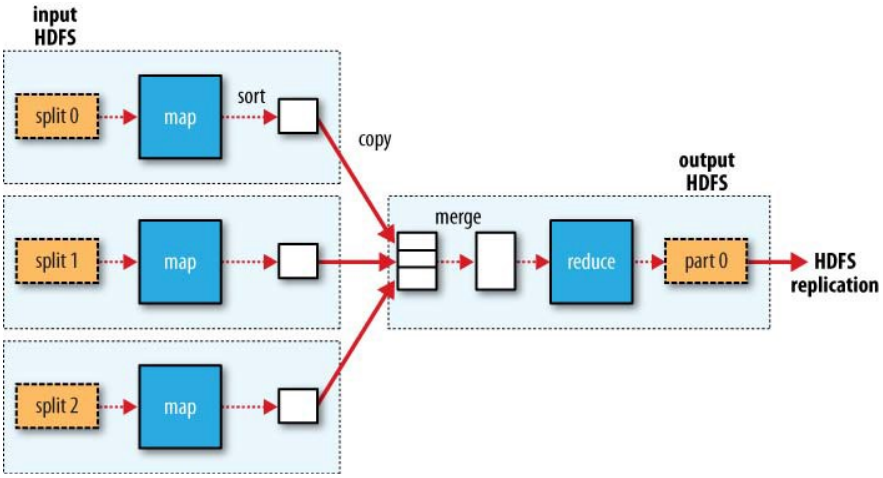

Figure 2-2. MapReduce data flow with a single reduce task

The number of reduce tasks is not governed by the size of the input, but is specified independently. In "The Default MapReduce Job" , you will see how to choose the number of reduce tasks for a given job. When there are multiple reducers, the map tasks *partition* their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general cases of multiple reduce tasks is illustrated in Figure 2-3. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time, as you will see in "Shuffle and Sort".

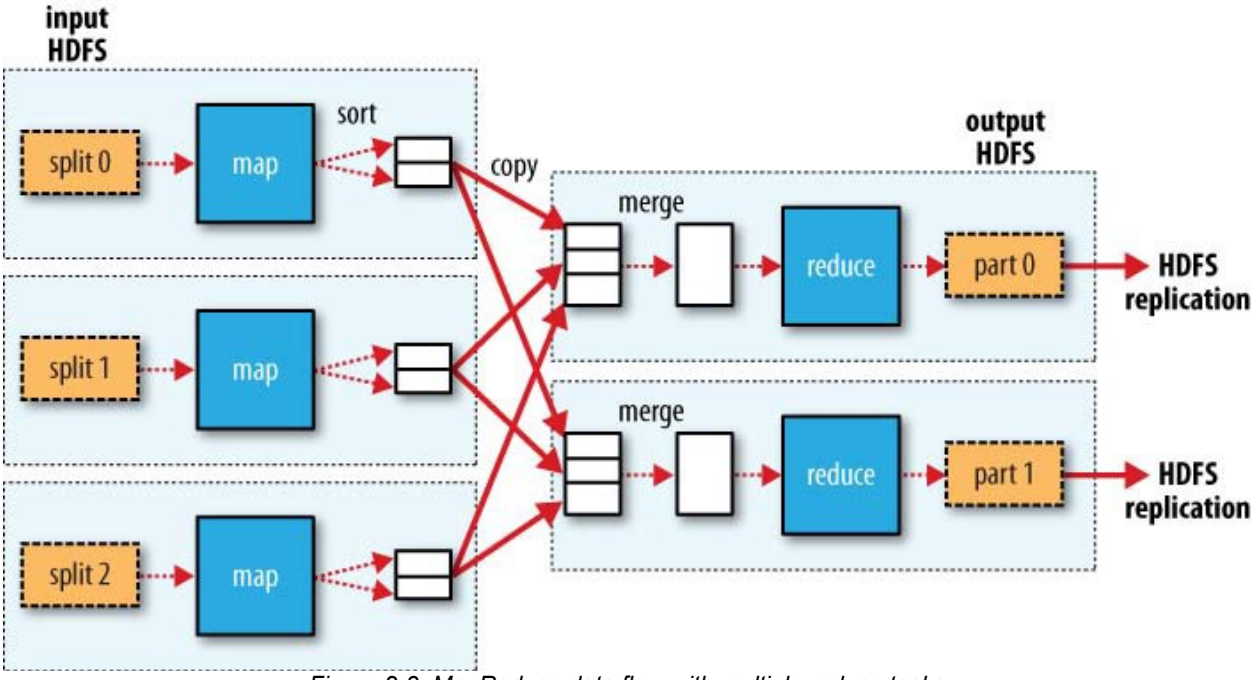

Figure 2-3. MapReduce data flow with multiple reduce tasks

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle since the processing can be carried out entirely in parallel. In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-4).

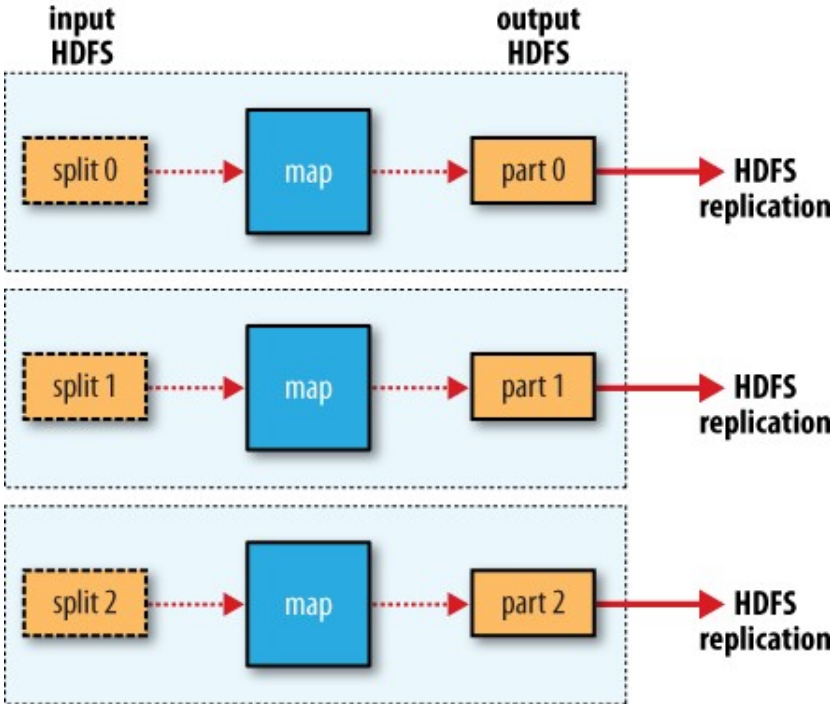

Figure 2-4. MapReduce data flow with no reduce tasks

Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a *combiner function* to be run on the map output—the combiner function's output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.

The contract for the combiner function constrains the type of function that may be used. This is best illustrated with an example. Suppose that for the maximum temperature example, readings for the year 1950 were processed by two maps (because they were in different splits). Imagine the first map produced the output:
(1950, 0)
(1950, 20)
(1950, 10)
And the second produced:
(1950, 25)
(1950, 15)
The reduce function would be called with a list of all the values:

(1950, [0, 20, 10, 25, 15])
with output:
(1950, 25)
since 25 is the maximum value in the list. We could use a combiner function that, just like the reduce function, finds the maximum temperature for each map output. The reduce would then be called with:
(1950, [20, 25])
and the reduce would produce the same output as before. More succinctly, we may express the function calls on the temperature values in this case as follows:
max(0, 20, 10, 25, 15) = *max*(*max*(0, 20, 10), *max*(25, 15)) = *max*(20, 25) = 25

Not all functions possess this property. For example, if we were calculating mean temperatures, then we couldn't use the mean as our combiner function, since:
mean(0, 20, 10, 25, 15) = 14
but:
mean(*mean*(0, 20, 10), *mean*(25, 15)) = *mean*(10, 20) = 15
The combiner function doesn't replace the reduce function. (How could it? The reduce function is still needed to process records with the same key from different maps.) But it can help cut down the amount of data shuffled between the maps and the reduces, and for this reason alone it is always worth considering whether you can use a combiner function in your MapReduce job.

Specifying a combiner function

Going back to the Java MapReduce program, the combiner function is defined using the Reducer interface, and for this application, it is the same implementation as the reducer function in MaxTemperatureReducer. The only change we need to make is to set the combiner class on the JobConf (see Example 2-7).

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

```
public class MaxTemperatureWithCombiner {
public static void main(String[] args) throws IOException {
if (args.length != 2) {
System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
"<output path>");
System.exit(-1);
}
JobConf conf = new JobConf(MaxTemperatureWithCombiner.class);
conf.setJobName("Max temperature");
```

```
FileInputFormat.addInputPath(conf, new Path(args[0]));
FileOutputFormat.setOutputPath(conf, new Path(args[1]));
conf.setMapperClass(MaxTemperatureMapper.class);
conf.setCombinerClass(MaxTemperatureReducer.class);
conf.setReducerClass(MaxTemperatureReducer.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(IntWritable.class);
JobClient.runJob(conf);
}
```

Running a Distributed MapReduce Job

The same program will run, without alteration, on a full dataset. This is the point of MapReduce: it scales to the size of your data and the size of your hardware. Here's one data point: on a 10-node EC2(Elastic Compute Cloud) cluster running High-CPU Extra Large Instances, the program took six minutes to run.

Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. *Hadoop Streaming* uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

Streaming is naturally suited for text processing (although, as of version 0.21.0, it can handle binary streams, too), and when used in text mode, it has a line-oriented view of data. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the framework guarantees are sorted by key, and writes its results to standard output. Let's illustrate this by rewriting our MapReduce program for finding maximum temperatures by year in Streaming.

Ruby

The map function can be expressed in Ruby as shown in Example 2-8.

Example 2-8. Map function for maximum temperature in Ruby

```
#!/usr/bin/env ruby
STDIN.each_line do |line|
val = line
year, temp, q = val[15,4], val[87,5], val[92,1]
```

```
puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

The program iterates over lines from standard input by executing a block for each line from STDIN (a global constant of type IO). The block pulls out the relevant fields from each input line, and, if the temperature is valid, writes the year and the temperature separated by a tab character \t to standard output (using puts).

Since the script just operates on standard input and output, it's trivial to test the script

without using Hadoop, simply using Unix pipes:

```
% cat input/ncdc/sample.txt |
ch02/src/main/ruby/max_temperature_map.rb
1950 +0000
1950 +0022
1950 -0011
1949 +0111
1949 +0078
```

The reduce function shown in Example 2-9 is a little more complex.

Example 2-9. Reduce function for maximum temperature in Ruby

```
#!/usr/bin/env ruby
last_key, max_val = nil, 0
STDIN.each_line do |line|
key, val = line.split("\t")
if last_key && last_key != key
puts "#{last_key}\t#{max_val}"
last_key, max_val = key, val.to_i
else
last_key, max_val = key, [max_val, val.to_i].max
end
end
puts "#{last_key}\t#{max_val}" if last_key
```

Again, the program iterates over lines from standard input, but this time we have to store some state as we process each key group. In this case, the keys are weather station identifiers, and we store the last key seen and the maximum temperature seen so far for that key. The MapReduce framework ensures that the keys are ordered, so we know that if a key is different from the previous one, we have moved into a new key group.

In contrast to the Java API, where you are provided an iterator over each key group, in Streaming you have to find key group boundaries in your program.

For each line, we pull out the key and value, then if we've just finished a group (last_key && last_key != key), we write the key and the maximum temperature for that group, separated by a tab character, before resetting

the maximum temperature for the new key. If we haven't just finished a group, we just update the maximum temperature for the current key.
The last line of the program ensures that a line is written for the last key group in the input.
We can now simulate the whole MapReduce pipeline with a Unix pipeline (which is equivalent to the Unix pipeline shown in Figure 2-1):

```
% cat input/ncdc/sample.txt | \
ch02/src/main/ruby/max_temperature_map.rb | \
sort | ch02/src/main/ruby/max_temperature_reduce.rb
1949 111
1950 22
```

The output is the same as the Java program, so the next step is to run it using Hadoop itself.
The hadoop command doesn't support a Streaming option; instead, you specify the Streaming JAR file along with the jar option. Options to the Streaming program specify the input and output paths, and the map and reduce scripts. This is what it looks like:

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-streaming.jar \
-input input/ncdc/sample.txt \
-output output \
-mapper ch02/src/main/ruby/max_temperature_map.rb \
-reducer ch02/src/main/ruby/max_temperature_reduce.rb
```

When running on a large dataset on a cluster, we should set the combiner, using the -combiner option.
From release 0.21.0, the combiner can be any Streaming command. For earlier releases, the combiner had to be written in Java, so as a workaround it was common to do manual combining in the mapper, without having to resort to Java. In this case, we could change the mapper to be a pipeline:

```
% hadoop jar $HADOOP_INSTALL/contrib/streaming/hadoop-*-streaming.jar \
-input input/ncdc/all \
-output output \
-mapper "ch02/src/main/ruby/max_temperature_map.rb | sort | \
ch02/src/main/ruby/max_temperature_reduce.rb" \
-reducer ch02/src/main/ruby/max_temperature_reduce.rb \
-file ch02/src/main/ruby/max_temperature_map.rb \
-file ch02/src/main/ruby/max_temperature_reduce.rb
```

Note also the use of -file, which we use when running Streaming programs on the cluster to ship the scripts to the cluster.
Python Streaming supports any programming language that can read from standard input, and write to standard output, so for readers more familiar

with Python, here's the same example again. The map script is in Example 2-10, and the reduce script is in Example2-11.

Example 2-10. Map function for maximum temperature in Python

```
#!/usr/bin/env python
import re
import sys
for line in sys.stdin:
val = line.strip()
(year, temp, q) = (val[15:19], val[87:92], val[92:93])
if (temp != "+9999" and re.match("[01459]", q)):
print "%s\t%s" % (year, temp)
```

Example 2-11. Reduce function for maximum temperature in Python

```
#!/usr/bin/env python
import sys
(last_key, max_val) = (None, 0)
for line in sys.stdin:
(key, val) = line.strip().split("\t")
if last_key and last_key != key:
print "%s\t%s" % (last_key, max_val)
(last_key, max_val) = (key, int(val))
else:
(last_key, max_val) = (key, max(max_val, int(val)))
if last_key:
print "%s\t%s" % (last_key, max_val)
```

We can test the programs and run the job in the same way we did in Ruby. For example,

to run a test:

```
% cat input/ncdc/sample.txt | ch02/src/main/python/max_temperature_map.py | \
sort | ch02/src/main/python/max_temperature_reduce.py
1949 111
1950 22
```