MapReduce

A Weather Dataset

For our example, we will write a program that mines weather data. Weather sensors collect data every hour at many locations across the globe and gather a large volume of log data which is a good candidate for analysis with MapReduce because we want to process all the data, and the data is semi-structured and record-oriented.

Data Format

The data we will use is from the National Climatic Data Center, or NCDC. The data i stored using a line-oriented ASCII format, in which each line is a record. The forma supports a rich set of meteorological elements, many of which are optional or wit variable data lengths. For simplicity, we focus on the basic elements, such as temperature which are always present and are of fixed width.

Below Example 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: Format of a National Climate Data Centre 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 zipped 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 pre-processed so that each.

Analyzing the Data with Unix Tools

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.

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

Example : 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 is 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 InstanceTo speed up the processing, we need to run parts of the program in parallel. In theory, this is straightforward: we could 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. In this case, 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. For this example, data for a particular year will typically be split into several chunks, each processed independently.We'll end up with the maximum temperature for each chunk, so the final step is to look for the highest of these maximums, for each year.

Third, you are still limited by the processing capacity of a single machine. If the best time you can achieve is 20 minutes with the number of processors you 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 is a great help.

Scaling Out

You've seen how MapReduce works for small inputs; now it's time to take a bird's-eye view of the system and look at the data flow for large inputs. For simplicity, the examples so far have used files on the local filesystem. However, to scale out, we need to store the data in a distributed filesystem (typically HDFS, which you'll learn about in the next chapter). This allows Hadoop to move the MapReduce computation to each machine hosting a part of the data, using Hadoop's resource management system, called YARN (see Chapter 4). Let's see how this works.

Analyzing Data with Hadoop

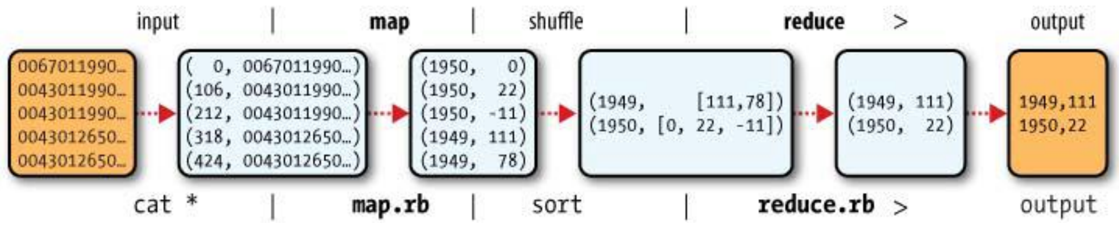
While the MapReduce programming model is at the heart of Hadoop, it is low-level and as such becomes a unproductive way for developers to write complex analysis jobs. To increase developer productivity, several higher-level languages and APIs have been created that abstract away the low-level details of the MapReduce programming model. There are several choices available for writing data analysis jobs. The Hive and Pig projects are popular choices that provide SQL-like and procedural data flow-like languages, respectively. HBase is also a popular way to store and analyze data in HDFS. It is a column-oriented database, and unlike MapReduce, provides random read and write access to data with low latency. MapReduce jobs can read and write data in HBase’s table format, but data processing is often done via HBase’s own client API. In this chapter, we will show how to use Spring for Apache Hadoop to write Java applications that use these Hadoop technologies.

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, but as we have no need for this, we ignore it. Our map function is simple. We 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, 006701199099999**1950**051507004...9999999N9+**0000**1+99999999999...)
(106, 004301199099999**1950**051512004...9999999N9+**0022**1+99999999999...)
(212, 004301199099999**1950**051518004...9999999N9**-0011**1+99999999999...)
(318, 004301265099999**1949**032412004...0500001N9+**0111**1+99999999999...)
(424, 004301265099999**1949**032418004...0500001N9+**0078**1+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



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, but as we have no need for this, we ignore it.

Our map function is simple. We pull out the year and the air temperature, because 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 reduce 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.

HOW MAPREDUCE WORKS

Hadoop MapReduce jobs are divided into a set of map tasks and reduce tasks that run in a distributed fashion on a cluster of computers. Each task work on a small subset of the data it has been assigned so that the load is spread across the cluster.

The input to a MapReduce job is a set of files in the data store that are spread out over the HDFS. In Hadoop, these files are split with an input format, which defines how to separate a files into input split. You can assume that input split is a byte-oriented view of a chunk of the files to be loaded by a map task.

The map task generally performs loading, parsing, transformation and filtering operations, whereas reduce task is responsible for grouping and aggregating the data produced by map tasks to generate final output. This is the way a wide range of problems can be solved with such a straightforward paradigm, from simple numerical aggregation to complex join operations and cartesian products.

Each map task in Hadoop is broken into following phases: record reader, mapper, combiner, partitioner. The output of map phase, called intermediate key and values are sent to the reducers. The reduce tasks are broken into following phases: shuffle, sort, reducer and output format. The map tasks are assigned by Hadoop framework to those DataNodes where the actual data to be processed resides. This ensures that the data typically doesn't have to move over the network to save the network bandwidth and data is computed on the local machine itself so called map task is data local.

Mapper

Record Reader:

The record reader translates an input split generated by input format into records. The purpose of record reader is to parse the data into record but doesn’t parse the record itself. It passes the data to the mapper in form of key/value pair. Usually the key in this context is positional information and the value is a chunk of data that composes a record. In our future articles we will discuss more about NLineInputFormat and custom record readers.

Map:

Map function is the heart of mapper task, which is executed on each key/value pair from the record reader to produce zero or more key/value pair, called intermediate pairs. The decision of what is key/value pair depends on what the MapReduce job is accomplishing. The data is grouped on key and the value is the information pertinent to the analysis in the reducer.

Combiner:

Its an optional component but highly useful and provides extreme performance gain of MapReduce job without any downside. Combiner is not applicable to all the MapReduce algorithms but where ever it can be applied it is always recommended to use. It takes the intermediate keys from the mapper and applies a user-provided method to aggregate values in a small scope of that one mapper. e.g sending (hadoop, 3) requires fewer bytes than sending (hadoop, 1) three times over the network. We will cover combiner in much more depth in our future articles.

Partitioner:

The partitioner takes the intermediate key/value pairs from mapper and split them into shards, one shard per reducer. This randomly distributes the keyspace evenly over the reducer, but still ensures that keys with the same value in different mappers end up at the same reducer. The partitioned data is written to the local filesystem for each map task and waits to be pulled by its respective reducer.

Reducer

Shuffle and Sort:

The reduce task start with the shuffle and sort step. This step takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running. These individual data pipes are then sorted by keys into one larger data list. The purpose of this sort is to group equivalent keys together so that their values can be iterated over easily in the reduce task.

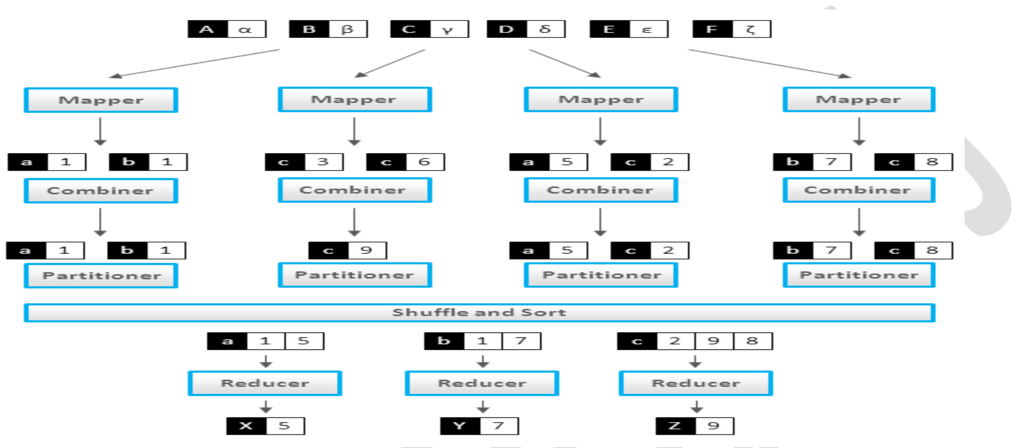
Reduce:

The reducer takes the grouped data as input and runs a reduce function once per key grouping. The function is passed the key and an iterator over all the values associated with that key. A wide range of processing can happen in this function, the data can be aggregated, filtered, and combined in a number of ways. Once it is done, it sends zero or more key/value pair to the final step, the output format.

Output Format:

The output format translate the final key/value pair from the reduce function and writes it out to a file by a record writer. By default, it will separate the key and value with a tab and separate record with a new line character. We will discuss in our future articles about how to write your own customized output format.

Following diagram shows the example for map Reduce execution.

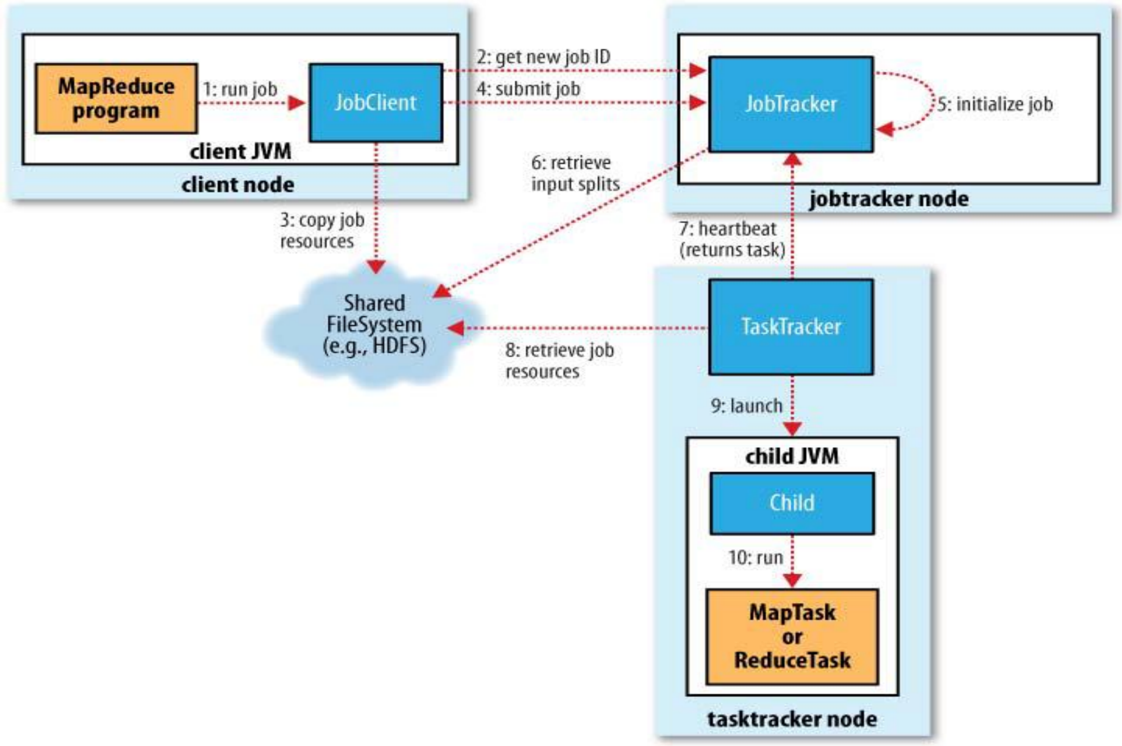


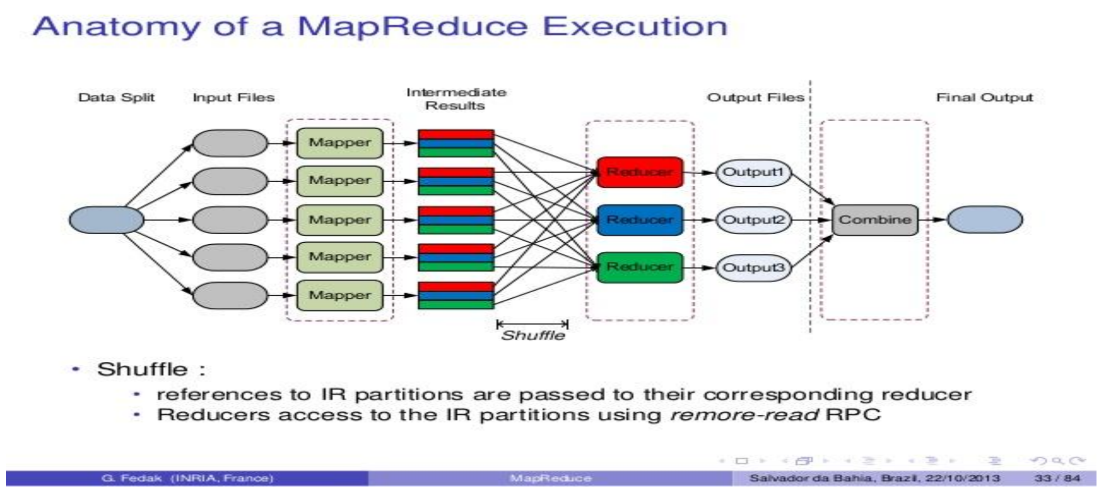
Anatomy of a MapReduce Job Run
You can run a MapReduce job with a single line of code: JobClient.runJob(conf). It's very short, but it conceals a great deal of processing behind the scenes. This section uncovers the steps Hadoop takes to run a job. The whole process is illustrated in Figure bellow. At the highest level, there are four independent entities:
• The client, which submits the MapReduce job.
• The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.
• The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.
• The distributed filesystem

The job submission process implemented by JobClient's submitJob() method does the following:

• Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker) (step2).
• Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
• Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program.
• Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the

mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).





Anatomy of a MapReduce Execution

- Shuffle :
  - references to IR partitions are passed to their corresponding reducer
  - Reducers access to the IR partitions using *remore-read* RPC

G. Fedak (INRIA, France) MapReduce Salvador da Bahia, Brazil, 22/10/2013 33 / 84

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.
Reducer function:

Reducer for maximum temperature example

```
import java.io.IOException;
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));
}
}
```

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function. The input types of the reduce function must match the output types of the map function: Text and IntWritable. And in this case, the output types of the reduce function are Text and IntWritable, for a year and its maximum. temperature, which we find by iterating through the temperatures and comparing each with a record of the highest found so far.The third piece of code runs the MapReduce job (see Example bellow).

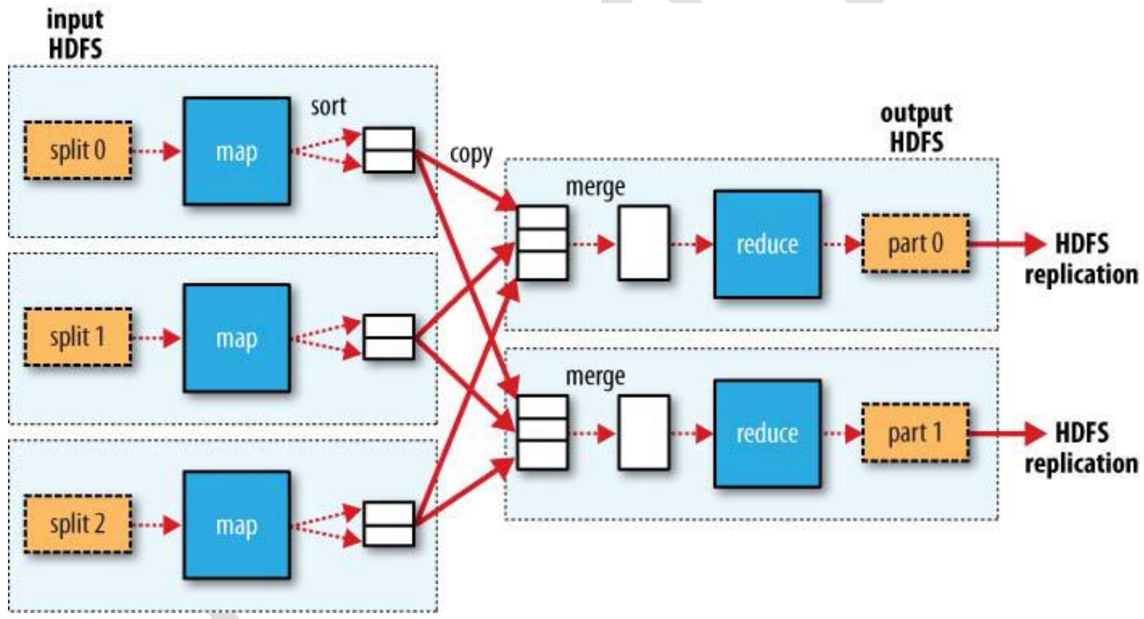
Example 2-5. Application to find the maximum temperature in the weather dataset

```
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);
}
}
```

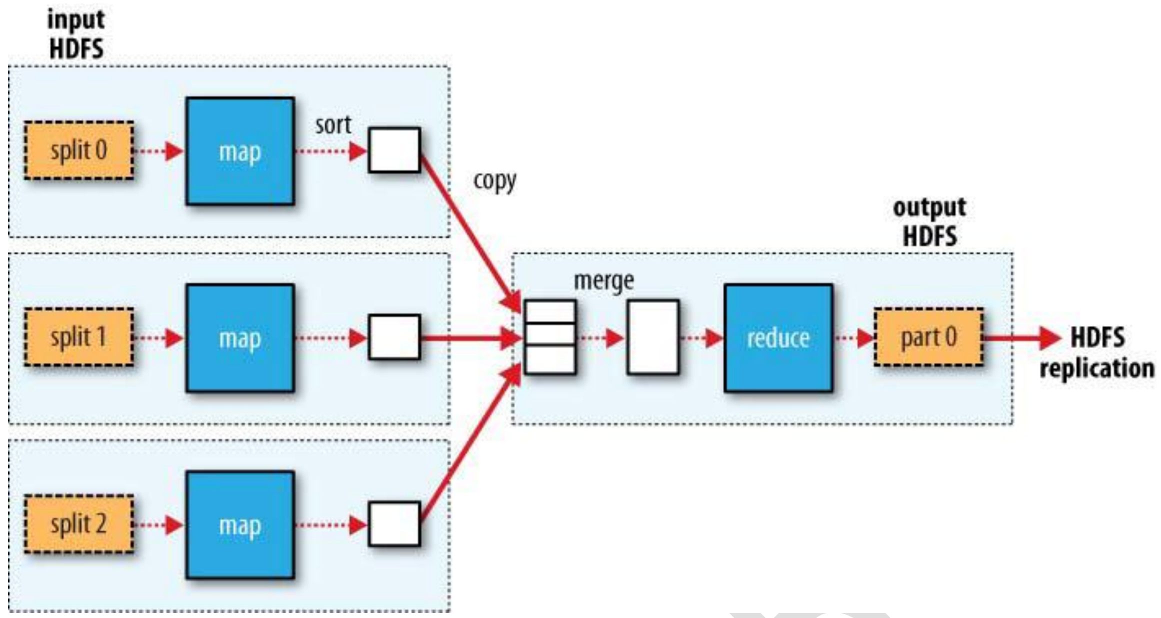
A JobConf object forms the specification of the job. It gives you control over how the job is run. When we run this job on a Hadoop cluster, we will package the code into a JAR file (which Hadoop will distribute around the cluster). Rather than explicitly specify the name of the JAR file, we can pass 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.
Scaling Out
You've seen how MapReduce works for small inputs; now it's time to take a bird's-eye view of the system and look at the data flow for large inputs. For simplicity, the examples so far have used files on the local filesystem. However, to scale out, we need to store the data in a distributed filesystem, typically HDFS (which you'll learn about in the next chapter), to allow Hadoop to move the MapReduce computation to each machine hosting a part of the data.
Let's see how this works.
Data Flow
First, some terminology. 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. The whole data flow with a single reduce task is illustrated in Figure. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes. *MapReduce data flow with multiple reduce tasks is shown bellow.*



Map Reduce data flow with multiple single tasks is shown bellow.



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).

Example . 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);
}
}
```

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.

Job Scheduling
Early versions of Hadoop had a very simple approach to scheduling users' jobs: they ran in order of submission, using a FIFO scheduler. Typically, each job would use the whole cluster, so jobs had to wait their turn. Although a shared cluster offers great potential for offering large resources to many users, the problem of sharing resources. fairly between users requires a better scheduler. Production jobs need to complete in a timely manner, while allowing users who are making smaller ad hoc queries to get results back in a reasonable time.
Later on, the ability to set a job's priority was added, via the mapred.job.priority property or the setJobPriority() method on JobClient (both of which take one of the values VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW). When the job scheduler is choosing the next job to run, it selects one with the highest priority. However, with the FIFO scheduler, priorities do not support *preemption*, so a high-priority job can still be blocked by a long-running low priority job that started before the high-priority job was scheduled.
MapReduce in Hadoop comes with a choice of schedulers. The default is the original FIFO queue-based scheduler, and there are also multiuser schedulers called the Fair Scheduler and the Capacity Scheduler.

The Fair Scheduler
The Fair Scheduler aims to give every user a fair share of the cluster capacity over time. If a single job is running, it gets all of the cluster. As more jobs are submitted, free task slots are given to the jobs in such a way as to give each user a fair share of the cluster. A short job belonging to one user will complete in a reasonable time even while another user's long job is running, and the long job will still make progress. Jobs are placed in pools, and by default, each user gets their own pool. A user who submits more jobs than a second user will not get any more cluster resources than the second, on average. It is also possible to define custom pools with guaranteed minimum capacities defined in terms of the number of map and reduce slots, and to set weightings for each pool.
The Fair Scheduler supports preemption, so if a pool has not received its fair share for a certain period of time, then the scheduler will kill tasks in pools running over capacity in order to give the slots to the pool running under capacity. The Fair Scheduler is a "contrib" module. To enable it, place its JAR file on Hadoop's classpath, by copying it from Hadoop's *contrib/fairscheduler* directory to the *lib* directory. Then set the property to:mapred.jobtracker.taskScheduler org.apache.hadoop.mapred.
The Fair Scheduler will work without further configuration, but to take full advantage of its features and how to configure it (including its web interface), refer to README in the *src/contrib/fairscheduler* directory of the distribution.

The Capacity Scheduler

The Capacity Scheduler takes a slightly different approach to multiuser scheduling. A cluster is made up of a number of queues (like the Fair Scheduler's pools), which may be hierarchical (so a queue may be the child of another queue), and each queue has an allocated capacity. This is like the Fair Scheduler, except that within each queue, jobs are scheduled using FIFO scheduling (with priorities). In effect, the Capacity Scheduler allows users or organizations (defined using queues) to simulate a separate MapReduce cluster with FIFO scheduling for each user or organization. The Fair Scheduler, by contrast, (which actually also supports FIFO job scheduling within pools as an option, making it like the Capacity Scheduler) enforces fair sharing within each pool, so running jobs share the pool's resources.