When running in local (standalone) mode, the programs in this book all assume that you have set the `HADOOP_CLASSPATH` in this way. The commands should be run from the directory that the example code is installed in.

The output from running the job provides some useful information. For example, we can see that the job was given an ID of `job_local26392882_0001`, and it ran one map task and one reduce task (with the following IDs: `attempt_local26392882_0001_m_000000_0` and `attempt_local26392882_0001_r_000000_0`). Knowing the job and task IDs can be very useful when debugging MapReduce jobs.

The last section of the output, titled "Counters," shows the statistics that Hadoop generates for each job it runs. These are very useful for checking whether the amount of data processed is what you expected. For example, we can follow the number of records that went through the system: five map input records produced five map output records (since the mapper emitted one output record for each valid input record), then five reduce input records in two groups (one for each unique key) produced two reduce output records.

The output was written to the *output* directory, which contains one output file per reducer. The job had a single reducer, so we find a single file, named *part-r-00000*:

```
% cat output/part-r-00000
1949 111
1950 22
```

This result is the same as when we went through it by hand earlier. We interpret this as saying that the maximum temperature recorded in 1949 was 11.1°C, and in 1950 it was 2.2°C.

## 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.

### 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*. The tasks are scheduled using YARN and run on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node.

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 user-defined 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 when 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, the overhead of managing the splits and 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, which is 128 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, because it doesn't use valuable cluster bandwidth. This is called the *data locality optimization*. Sometimes, however, all the nodes hosting the HDFS block replicas for a map task's input split are running other map tasks, so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. The three possibilities are illustrated in Figure 2-2.

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.

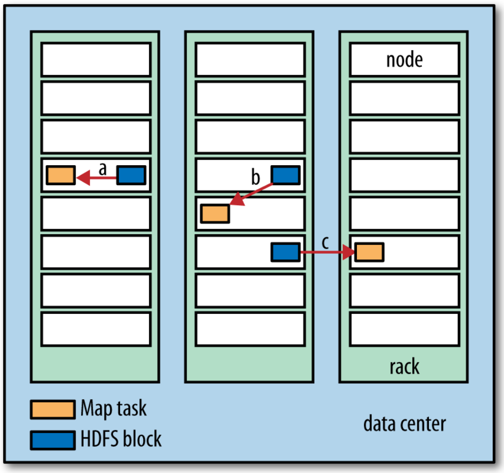

*Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks*

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. As explained in Chapter 3, 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 for reliability. 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 2-3. The dotted boxes indicate nodes, the dotted arrows show data transfers on a node, and the solid arrows show data transfers between nodes.

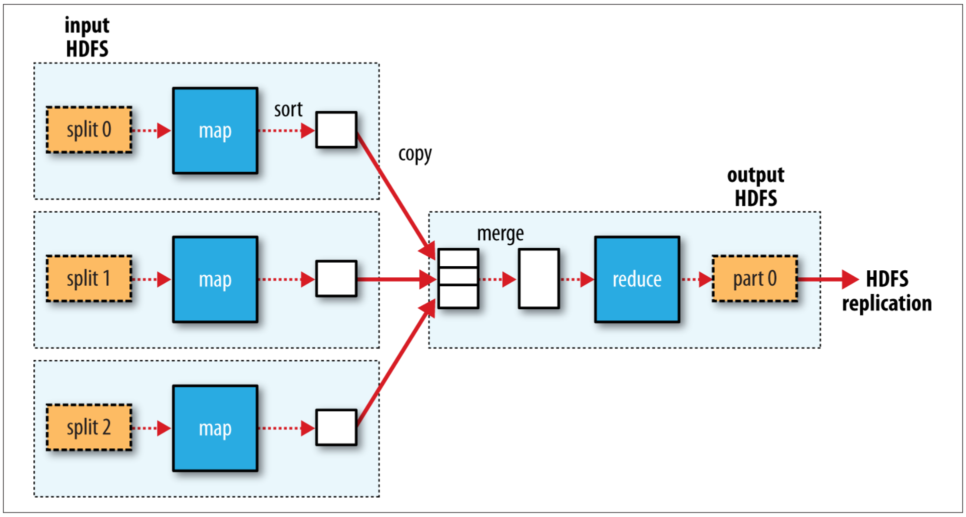

*Figure 2-3. MapReduce data flow with a single reduce task*

The number of reduce tasks is not governed by the size of the input, but instead is specified independently. In "The Default MapReduce Job" on page 214, 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 case of multiple reduce tasks is illustrated in Figure 2-4. 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" on page 197.

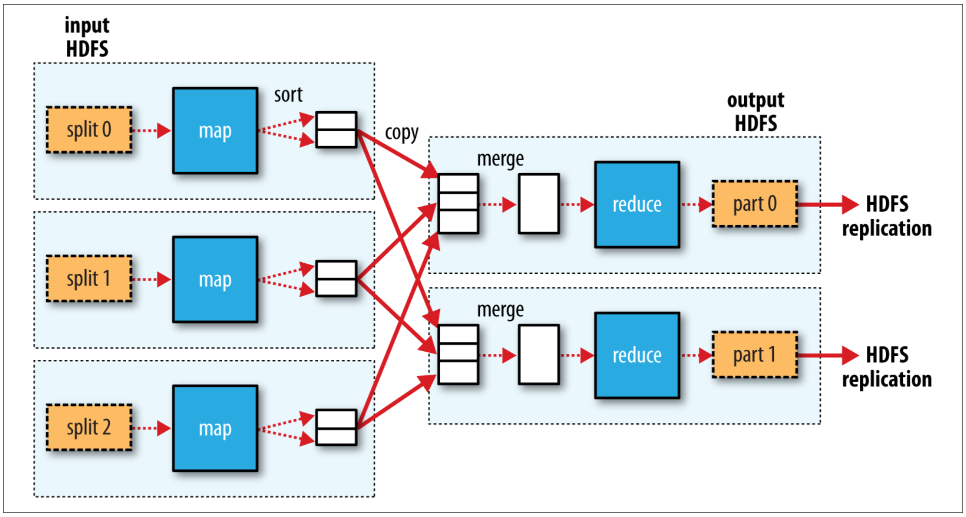

*Figure 2-4. MapReduce data flow with multiple reduce tasks*

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel (a few examples are discussed in "NLineInputFormat" on page 234). In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).

## 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, and the combiner function's output forms the input to the reduce function. Because 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.

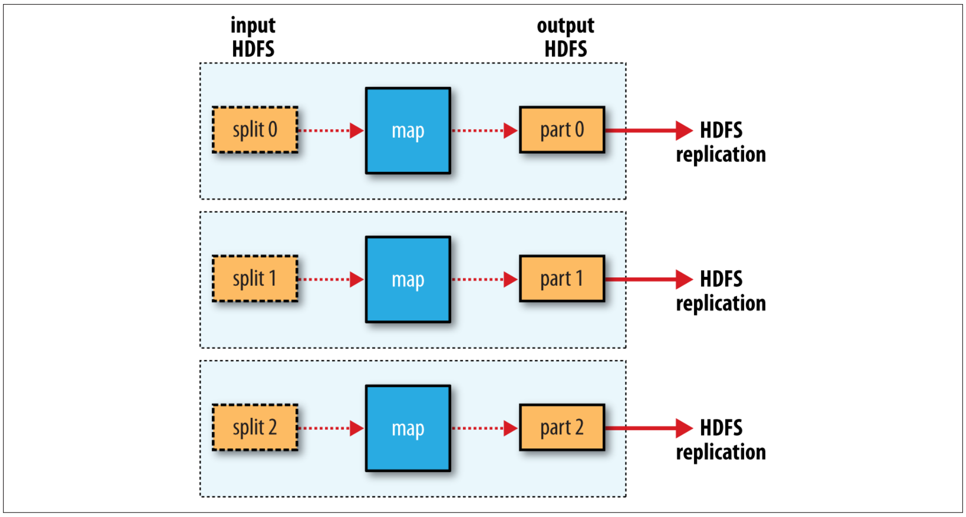

*Figure 2-5. MapReduce data flow with no reduce tasks*

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 function would then be called with:

```
(1950, [20, 25])
```

and 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.[1] For example, if we were calculating mean temperatures, we couldn't use the mean as our combiner function, because:

```
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 mappers and the reducers, 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` class, and for this application, it is the same implementation as the reduce function in `MaxTemperatureReducer`. The only change we need to make is to set the combiner class on the `Job` (see Example 2-6).

*Example 2-6. Application to find the maximum temperature, using a combiner function for efficiency*

```java
public class MaxTemperatureWithCombiner {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

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

1. Functions with this property are called *commutative* and *associative*. They are also sometimes referred to as *distributive*, such as by Jim Gray et al.'s "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," February1995.

```
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

### 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 cluster running High-CPU Extra Large instances, the program took six minutes to run.[2]

We'll go through the mechanics of running programs on a cluster in Chapter 6.

## 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.[3]

Streaming is naturally suited for text processing. 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 2-7.

2. This is a factor of seven faster than the serial run on one machine using *awk*. The main reason it wasn't proportionately faster is because the input data wasn't evenly partitioned. For convenience, the input files were gzipped by year, resulting in large files for later years in the dataset, when the number of weather records was much higher.

3. Hadoop Pipes is an alternative to Streaming for C++ programmers. It uses sockets to communicate with the process running the C++ map or reduce function.

*Example 2-7. 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`).

It's worth drawing out a design difference between Streaming and the Java MapReduce API. The Java API is geared toward processing your map function one record at a time. The framework calls the `map()` method on your `Mapper` for each record in the input, whereas with Streaming the map program can decide how to process the input—for example, it could easily read and process multiple lines at a time since it's in control of the reading. The user's Java map implementation is "pushed" records, but it's still possible to consider multiple lines at a time by accumulating previous lines in an instance variable in the `Mapper`.[4] In this case, you need to implement the `cleanup()` method so that you know when the last record has been read, so you can finish processing the last group of lines.

Because the script just operates on standard input and output, it's trivial to test the script without using Hadoop, simply by using Unix pipes:

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

The reduce function shown in Example 2-8 is a little more complex.

*Example 2-8. Reduce function for maximum temperature in Ruby*

```
#!/usr/bin/env ruby

last_key, max_val = nil, -1000000
STDIN.each_line do |line|
  key, val = line.split("\t")
```

4. Alternatively, you could use "pull"-style processing in the new MapReduce API; see Appendix D.

```
  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 the years, 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-mr-intro/src/main/ruby/max_temperature_map.rb | \
  sort | ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
1949	111
1950	22
```

The output is the same as that of 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_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -input input/ncdc/sample.txt \
  -output output \
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
```

When running on a large dataset on a cluster, we should use the `-combiner` option to set the combiner:

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
  -files ch02-mr-intro/src/main/ruby/max_temperature_map.rb,\
ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
  -input input/ncdc/all \
  -output output \
  -mapper ch02-mr-intro/src/main/ruby/max_temperature_map.rb \
  -combiner ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb \
  -reducer ch02-mr-intro/src/main/ruby/max_temperature_reduce.rb
```

Note also the use of `-files`, 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.[5] The map script is in Example 2-9, and the reduce script is in Example 2-10.

*Example 2-9. Map function for maximum temperature in Python*

```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 2-10. Reduce function for maximum temperature in Python*

```python
#!/usr/bin/env python

import sys

(last_key, max_val) = (None, -sys.maxint)
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)))
```

5. As an alternative to Streaming, Python programmers should consider Dumbo, which makes the Streaming MapReduce interface more Pythonic and easier to use.

```
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-mr-intro/src/main/python/max_temperature_map.py | \
  sort | ch02-mr-intro/src/main/python/max_temperature_reduce.py
1949    111
1950    22
```