CHAPTER 2

MapReduce

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages; in this chapter, we look at the same program expressed in Java, Ruby, and Python. Most importantly, 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, so let's start by looking at one.

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 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, we 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 annotated. 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 Climatic 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
C
N
010000   # visibility distance (meters)
1        # quality code
N
9
-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
```

Datafiles 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
```

There are tens of thousands of weather stations, so 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. (The means by which this was carried out is described in Appendix C.)

Analyzing the Data with Unix Tools

What's the highest recorded global temperature for each year in the dataset? We will answer this first without using Hadoop, as this information will provide a performance baseline and a useful means to check our results.

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 is turned into an integer by adding 0. Next, a test is applied to see whether the temperature is valid (the value 9999 signifies a missing value in the NCDC dataset) and whether 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, 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 require further processing. In this case, the result for each year is independent of other years, and they 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 into the categories of coordination and reliability. Who runs the overall job? How do we deal with failed processes?

So, although 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.

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, 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.

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 this chapter when we look at Hadoop Streaming.

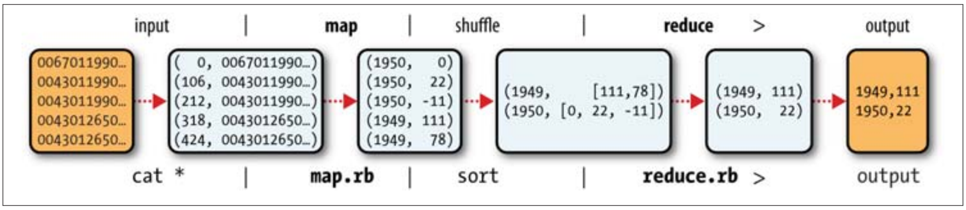

Figure 2-1. MapReduce logical data flow

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 the `Mapper` class, which declares an abstract `map()` method. Example 2-3 shows the implementation of our map function.

Example 2-3. Mapper for the maximum temperature example

```java
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.mapreduce.Mapper;

public class MaxTemperatureMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    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]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

The `Mapper` class 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 using 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 `Context` 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 the maximum temperature example

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(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 2-5).

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

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class MaxTemperature {

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

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

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

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

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

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

A `Job` object forms the specification of the job and 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 specifying the name of the JAR file, we can pass a class in the `Job`'s `setJarByClass()` method, which Hadoop will use to locate the relevant JAR file by looking for the JAR file containing this class.

Having constructed a `Job` 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 `setOutputPath()` method on `FileOutputFormat`. It specifies a directory where the output files from the reduce function are written. The directory shouldn't exist before running the job because 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 that of 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 reduce function, and must match what the `Reduce` class produces. The map output types default to the same types, so they do not need to be set if the mapper produces the same types as the reducer (as it does in our case). However, if they are different, the map output types must be set using the `setMapOutputKeyClass()` and `setMapOutputValueClass()` methods.

The input types are controlled via the input format, which we have not explicitly set because 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 `waitForCompletion()` method on `Job` submits the job and waits for it to finish. The single argument to the method is a flag indicating whether verbose output is generated. When `true`, the job writes information about its progress to the console.

The return value of the `waitForCompletion()` method is a Boolean indicating success (`true`) or failure (`false`), which we translate into the program's exit code of `0` or `1`.

> The Java MapReduce API used in this section, and throughout the book, is called the "new API"; it replaces the older, functionally equivalent API. The differences between the two APIs are explained in Appendix D, along with tips on how to convert between the two APIs. You can also find the old API equivalent of the maximum temperature application there.

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. Then, install and compile the examples using the instructions on the book's website.

Let's test it on the five-line sample discussed earlier (the output has been slightly reformatted to fit the page, and some lines have been removed):

```
% export HADOOP_CLASSPATH=hadoop-examples.jar
% hadoop MaxTemperature input/ncdc/sample.txt output
14/09/16 09:48:39 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
14/09/16 09:48:40 WARN mapreduce.JobSubmitter: Hadoop command-line option
parsing not performed. Implement the Tool interface and execute your application
with ToolRunner to remedy this.
14/09/16 09:48:40 INFO input.FileInputFormat: Total input paths to process : 1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: number of splits:1
14/09/16 09:48:40 INFO mapreduce.JobSubmitter: Submitting tokens for job:
job_local26392882_0001
14/09/16 09:48:40 INFO mapreduce.Job: The url to track the job:
http://localhost:8080/
14/09/16 09:48:40 INFO mapreduce.Job: Running job: job_local26392882_0001
14/09/16 09:48:40 INFO mapred.LocalJobRunner: OutputCommitter set in config null
14/09/16 09:48:40 INFO mapred.LocalJobRunner: OutputCommitter is
org.apache.hadoop.mapreduce.lib.output.FileOutputCommitter
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for map tasks
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:
attempt_local26392882_0001_m_000000_0
14/09/16 09:48:40 INFO mapred.Task:  Using ResourceCalculatorProcessTree : null
14/09/16 09:48:40 INFO mapred.LocalJobRunner:
14/09/16 09:48:40 INFO mapred.Task: Task:attempt_local26392882_0001_m_000000_0
is done. And is in the process of committing
14/09/16 09:48:40 INFO mapred.LocalJobRunner: map
14/09/16 09:48:40 INFO mapred.Task: Task 'attempt_local26392882_0001_m_000000_0'
 done.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Finishing task:
attempt_local26392882_0001_m_000000_0
14/09/16 09:48:40 INFO mapred.LocalJobRunner: map task executor complete.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Waiting for reduce tasks
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Starting task:
attempt_local26392882_0001_r_000000_0
14/09/16 09:48:40 INFO mapred.Task:  Using ResourceCalculatorProcessTree : null
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Merger: Merging 1 sorted segments
14/09/16 09:48:40 INFO mapred.Merger: Down to the last merge-pass, with 1
segments left of total size: 50 bytes
14/09/16 09:48:40 INFO mapred.Merger: Merging 1 sorted segments
14/09/16 09:48:40 INFO mapred.Merger: Down to the last merge-pass, with 1
segments left of total size: 50 bytes
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Task: Task:attempt_local26392882_0001_r_000000_0
is done. And is in the process of committing
14/09/16 09:48:40 INFO mapred.LocalJobRunner: 1 / 1 copied.
14/09/16 09:48:40 INFO mapred.Task: Task attempt_local26392882_0001_r_000000_0
```

```
is allowed to commit now
14/09/16 09:48:40 INFO output.FileOutputCommitter: Saved output of task
'attempt...local26392882_0001_r_000000_0' to file:/Users/tom/book-workspace/
hadoop-book/output/_temporary/0/task_local26392882_0001_r_000000
14/09/16 09:48:40 INFO mapred.LocalJobRunner: reduce > reduce
14/09/16 09:48:40 INFO mapred.Task: Task 'attempt_local26392882_0001_r_000000_0'
 done.
14/09/16 09:48:40 INFO mapred.LocalJobRunner: Finishing task:
attempt_local26392882_0001_r_000000_0
14/09/16 09:48:40 INFO mapred.LocalJobRunner: reduce task executor complete.
14/09/16 09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 running in uber
 mode : false
14/09/16 09:48:41 INFO mapreduce.Job:  map 100% reduce 100%
14/09/16 09:48:41 INFO mapreduce.Job: Job job_local26392882_0001 completed
successfully
14/09/16 09:48:41 INFO mapreduce.Job: Counters: 30
    File System Counters
        FILE: Number of bytes read=377168
        FILE: Number of bytes written=828464
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
    Map-Reduce Framework
        Map input records=5
        Map output records=5
        Map output bytes=45
        Map output materialized bytes=61
        Input split bytes=129
        Combine input records=0
        Combine output records=0
        Reduce input groups=2
        Reduce shuffle bytes=61
        Reduce input records=5
        Reduce output records=2
        Spilled Records=10
        Shuffled Maps =1
        Failed Shuffles=0
        Merged Map outputs=1
        GC time elapsed (ms)=39
        Total committed heap usage (bytes)=226754560
    File Input Format Counters
        Bytes Read=529
    File Output Format Counters
        Bytes Written=29
```

When the `hadoop` command is invoked with a classname as the first argument, it launches a Java virtual machine (JVM) to run the class. The `hadoop` command 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.

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.

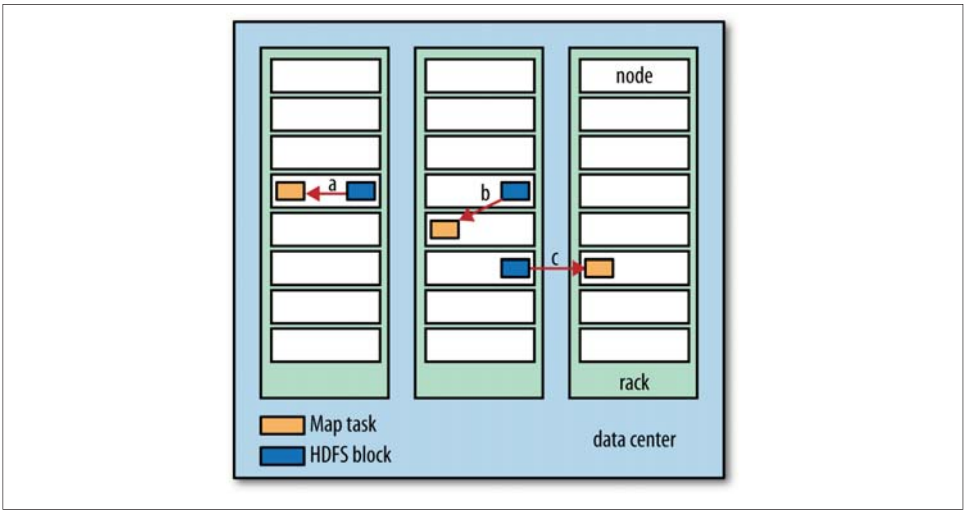

Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

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.

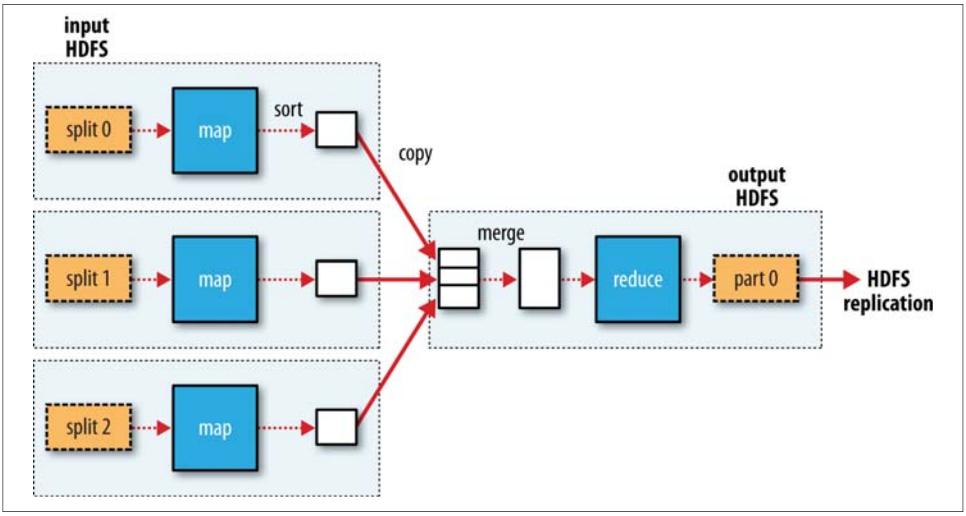

Figure 2-3. MapReduce data flow with a single reduce task

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.

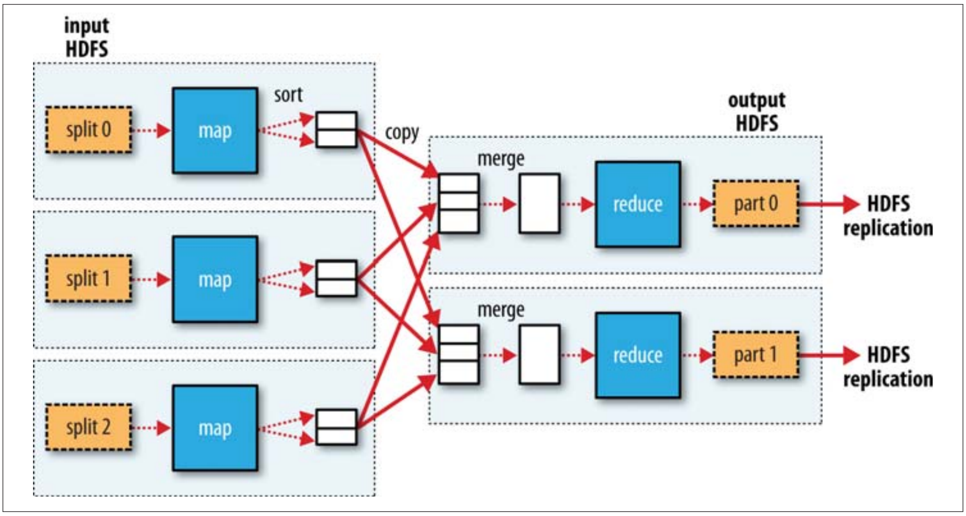

Figure 2-4. MapReduce data flow with multiple reduce tasks

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.

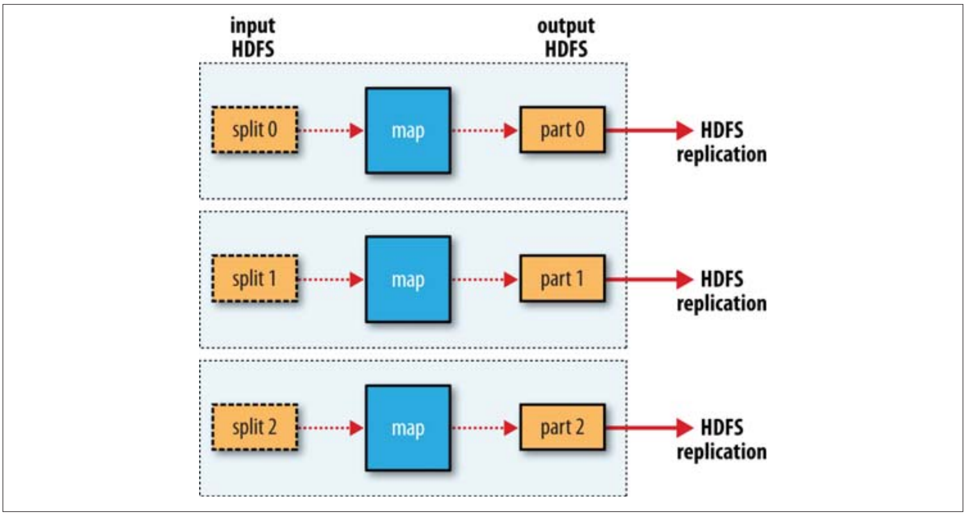

Figure 2-5. MapReduce data flow with no reduce tasks

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

```
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.

There are two properties that we set in the pseudodistributed configuration that deserve further explanation. The first is `fs.defaultFS`, set to `hdfs://localhost/`, which is used to set a default filesystem for Hadoop.[5] Filesystems are specified by a URI, and here we have used an `hdfs` URI to configure Hadoop to use HDFS by default. The HDFS daemons will use this property to determine the host and port for the HDFS namenode. We'll be running it on localhost, on the default HDFS port, 8020. And HDFS clients will use this property to work out where the namenode is running so they can connect to it.

We set the second property, `dfs.replication`, to 1 so that HDFS doesn't replicate filesystem blocks by the default factor of three. When running with a single datanode, HDFS can't replicate blocks to three datanodes, so it would perpetually warn about blocks being under-replicated. This setting solves that problem.

Basic Filesystem Operations

The filesystem is ready to be used, and we can do all of the usual filesystem operations, such as reading files, creating directories, moving files, deleting data, and listing directories. You can type `hadoop fs -help` to get detailed help on every command.

Start by copying a file from the local filesystem to HDFS:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt \
  hdfs://localhost/user/tom/quangle.txt
```

This command invokes Hadoop's filesystem shell command `fs`, which supports a number of subcommands—in this case, we are running `-copyFromLocal`. The local file *quangle.txt* is copied to the file */user/tom/quangle.txt* on the HDFS instance running on localhost. In fact, we could have omitted the scheme and host of the URI and picked up the default, `hdfs://localhost`, as specified in *core-site.xml*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt
```

We also could have used a relative path and copied the file to our home directory in HDFS, which in this case is */user/tom*:

```
% hadoop fs -copyFromLocal input/docs/quangle.txt quangle.txt
```

Let's copy the file back to the local filesystem and check whether it's the same:

```
% hadoop fs -copyToLocal quangle.txt quangle.copy.txt
% md5 input/docs/quangle.txt quangle.copy.txt
MD5 (input/docs/quangle.txt) = e7891a2627cf263a079fb0f18256ffb2
MD5 (quangle.copy.txt) = e7891a2627cf263a079fb0f18256ffb2
```

5. In Hadoop 1, the name for this property was `fs.default.name`. Hadoop 2 introduced many new property names, and deprecated the old ones (see "Which Properties Can I Set?" on page 150). This book uses the new property names.

The MD5 digests are the same, showing that the file survived its trip to HDFS and is back intact.

Finally, let's look at an HDFS file listing. We create a directory first just to see how it is displayed in the listing:

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x   - tom supergroup          0 2014-10-04 13:22 books
-rw-r--r--   1 tom supergroup        119 2014-10-04 13:21 quangle.txt
```

The information returned is very similar to that returned by the Unix command `ls -l`, with a few minor differences. The first column shows the file mode. The second column is the replication factor of the file (something a traditional Unix filesystem does not have). Remember we set the default replication factor in the site-wide configuration to be 1, which is why we see the same value here. The entry in this column is empty for directories because the concept of replication does not apply to them—directories are treated as metadata and stored by the namenode, not the datanodes. The third and fourth columns show the file owner and group. The fifth column is the size of the file in bytes, or zero for directories. The sixth and seventh columns are the last modified date and time. Finally, the eighth column is the name of the file or directory.

File Permissions in HDFS

HDFS has a permissions model for files and directories that is much like the POSIX model. There are three types of permission: the read permission (`r`), the write permission (`w`), and the execute permission (`x`). The read permission is required to read files or list the contents of a directory. The write permission is required to write a file or, for a directory, to create or delete files or directories in it. The execute permission is ignored for a file because you can't execute a file on HDFS (unlike POSIX), and for a directory this permission is required to access its children.

Each file and directory has an *owner*, a *group*, and a *mode*. The mode is made up of the permissions for the user who is the owner, the permissions for the users who are members of the group, and the permissions for users who are neither the owners nor members of the group.

By default, Hadoop runs with security disabled, which means that a client's identity is not authenticated. Because clients are remote, it is possible for a client to become an arbitrary user simply by creating an account of that name on the remote system. This is not possible if security is turned on; see "Security" on page 309. Either way, it is worthwhile having permissions enabled (as they are by default; see the `dfs.permissions.enabled` property) to avoid accidental modification or deletion of substantial parts of the filesystem, either by users or by automated tools or programs.

When permissions checking is enabled, the owner permissions are checked if the client's username matches the owner, and the group permissions are checked if the client is a member of the group; otherwise, the other permissions are checked.

There is a concept of a superuser, which is the identity of the namenode process. Permissions checks are not performed for the superuser.

Hadoop Filesystems

Hadoop has an abstract notion of filesystems, of which HDFS is just one implementation. The Java abstract class `org.apache.hadoop.fs.FileSystem` represents the client interface to a filesystem in Hadoop, and there are several concrete implementations. The main ones that ship with Hadoop are described in Table 3-1.

Table 3-1. Hadoop filesystems

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	`file`	`fs.LocalFileSystem`	A filesystem for a locally connected disk with client-side checksums. Use `RawLocalFileSystem` for a local filesystem with no checksums. See "LocalFileSystem" on page 99.
HDFS	`hdfs`	`hdfs.DistributedFileSystem`	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
WebHDFS	`webhdfs`	`hdfs.web.WebHdfsFileSystem`	A filesystem providing authenticated read/write access to HDFS over HTTP. See "HTTP" on page 54.
Secure WebHDFS	`swebhdfs`	`hdfs.web.SWebHdfsFileSystem`	The HTTPS version of WebHDFS.
HAR	`har`	`fs.HarFileSystem`	A filesystem layered on another filesystem for archiving files. Hadoop Archives are used for packing lots of files in HDFS into a single archive file to reduce the namenode's memory usage. Use the `hadoop archive` command to create HAR files.
View	`viewfs`	`viewfs.ViewFileSystem`	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see "HDFS Federation" on page 48).
FTP	`ftp`	`fs.ftp.FTPFileSystem`	A filesystem backed by an FTP server.
S3	`s3a`	`fs.s3a.S3AFileSystem`	A filesystem backed by Amazon S3. Replaces the older `s3n` (S3 native) implementation.

CHAPTER 4
YARN

Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system. YARN was introduced in Hadoop 2 to improve the MapReduce implementation, but it is general enough to support other distributed computing paradigms as well.

YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code. Instead, users write to higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN and hide the resource management details from the user. The situation is illustrated in Figure 4-1, which shows some distributed computing frameworks (MapReduce, Spark, and so on) running as *YARN applications* on the cluster compute layer (YARN) and the cluster storage layer (HDFS and HBase).

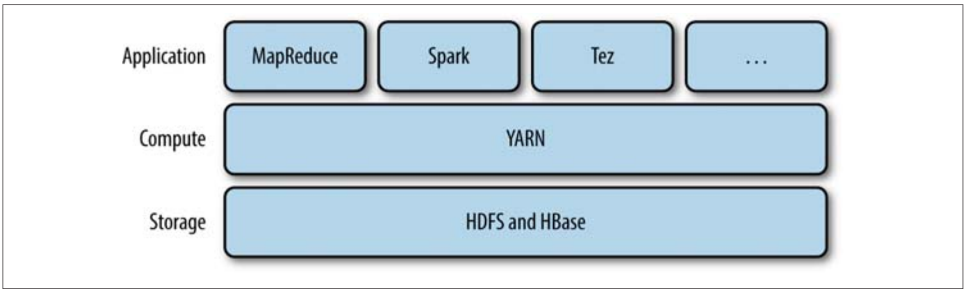

Figure 4-1. YARN applications

There is also a layer of applications that build on the frameworks shown in Figure 4-1. Pig, Hive, and Crunch are all examples of processing frameworks that run on MapReduce, Spark, or Tez (or on all three), and don't interact with YARN directly.

This chapter walks through the features in YARN and provides a basis for understanding later chapters in Part IV that cover Hadoop's distributed processing frameworks.

Anatomy of a YARN Application Run

YARN provides its core services via two types of long-running daemon: a *resource manager* (one per cluster) to manage the use of resources across the cluster, and *node managers* running on all the nodes in the cluster to launch and monitor *containers*. A container executes an application-specific process with a constrained set of resources (memory, CPU, and so on). Depending on how YARN is configured (see "YARN" on page 300), a container may be a Unix process or a Linux cgroup. Figure 4-2 illustrates how YARN runs an application.

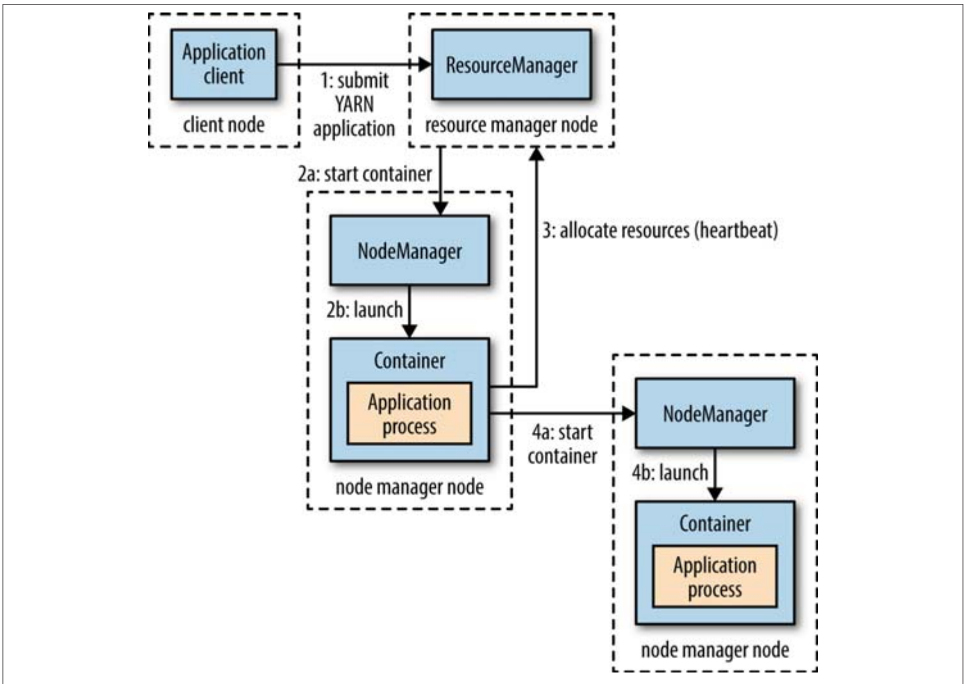

Figure 4-2. How YARN runs an application

To run an application on YARN, a client contacts the resource manager and asks it to run an *application master* process (step 1 in Figure 4-2). The resource manager then finds a node manager that can launch the application master in a container (steps 2a and 2b).[1] Precisely what the application master does once it is running depends on the application. It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the resource managers (step 3), and use them to run a distributed computation (steps 4a and 4b). The latter is what the MapReduce YARN application does, which we'll look at in more detail in "Anatomy of a MapReduce Job Run" on page 185.

Notice from Figure 4-2 that YARN itself does not provide any way for the parts of the application (client, master, process) to communicate with one another. Most nontrivial YARN applications use some form of remote communication (such as Hadoop's RPC layer) to pass status updates and results back to the client, but these are specific to the application.

Resource Requests

YARN has a flexible model for making resource requests. A request for a set of containers can express the amount of computer resources required for each container (memory and CPU), as well as locality constraints for the containers in that request.

Locality is critical in ensuring that distributed data processing algorithms use the cluster bandwidth efficiently,[2] so YARN allows an application to specify locality constraints for the containers it is requesting. Locality constraints can be used to request a container on a specific node or rack, or anywhere on the cluster (off-rack).

Sometimes the locality constraint cannot be met, in which case either no allocation is made or, optionally, the constraint can be loosened. For example, if a specific node was requested but it is not possible to start a container on it (because other containers are running on it), then YARN will try to start a container on a node in the same rack, or, if that's not possible, on any node in the cluster.

In the common case of launching a container to process an HDFS block (to run a map task in MapReduce, say), the application will request a container on one of the nodes hosting the block's three replicas, or on a node in one of the racks hosting the replicas, or, failing that, on any node in the cluster.

A YARN application can make resource requests at any time while it is running. For example, an application can make all of its requests up front, or it can take a more dynamic approach whereby it requests more resources dynamically to meet the changing needs of the application.

1. It's also possible for the client to start the application master, possibly outside the cluster, or in the same JVM as the client. This is called an *unmanaged application master*.

2. For more on this topic see "Scaling Out" on page 30 and "Network Topology and Hadoop" on page 70.

Spark takes the first approach, starting a fixed number of executors on the cluster (see "Spark on YARN" on page 571). MapReduce, on the other hand, has two phases: the map task containers are requested up front, but the reduce task containers are not started until later. Also, if any tasks fail, additional containers will be requested so the failed tasks can be rerun.

Application Lifespan

The lifespan of a YARN application can vary dramatically: from a short-lived application of a few seconds to a long-running application that runs for days or even months. Rather than look at how long the application runs for, it's useful to categorize applications in terms of how they map to the jobs that users run. The simplest case is one application per user job, which is the approach that MapReduce takes.

The second model is to run one application per workflow or user session of (possibly unrelated) jobs. This approach can be more efficient than the first, since containers can be reused between jobs, and there is also the potential to cache intermediate data between jobs. Spark is an example that uses this model.

The third model is a long-running application that is shared by different users. Such an application often acts in some kind of coordination role. For example, Apache Slider has a long-running application master for launching other applications on the cluster. This approach is also used by Impala (see "SQL-on-Hadoop Alternatives" on page 484) to provide a proxy application that the Impala daemons communicate with to request cluster resources. The "always on" application master means that users have very low-latency responses to their queries since the overhead of starting a new application master is avoided.[3]

Building YARN Applications

Writing a YARN application from scratch is fairly involved, but in many cases is not necessary, as it is often possible to use an existing application that fits the bill. For example, if you are interested in running a directed acyclic graph (DAG) of jobs, then Spark or Tez is appropriate; or for stream processing, Spark, Samza, or Storm works.[4]

There are a couple of projects that simplify the process of building a YARN application. Apache Slider, mentioned earlier, makes it possible to run existing distributed applications on YARN. Users can run their own instances of an application (such as HBase) on a cluster, independently of other users, which means that different users can run different versions of the same application. Slider provides controls to change the number

3. The low-latency application master code lives in the Llama project.

4. All of these projects are Apache Software Foundation projects.

of nodes an application is running on, and to suspend then resume a running application.

Apache Twill is similar to Slider, but in addition provides a simple programming model for developing distributed applications on YARN. Twill allows you to define cluster processes as an extension of a Java `Runnable`, then runs them in YARN containers on the cluster. Twill also provides support for, among other things, real-time logging (log events from runnables are streamed back to the client) and command messages (sent from the client to runnables).

In cases where none of these options are sufficient—such as an application that has complex scheduling requirements—then the *distributed shell* application that is a part of the YARN project itself serves as an example of how to write a YARN application. It demonstrates how to use YARN's client APIs to handle communication between the client or application master and the YARN daemons.

YARN Compared to MapReduce 1

The distributed implementation of MapReduce in the original version of Hadoop (version 1 and earlier) is sometimes referred to as "MapReduce 1" to distinguish it from MapReduce 2, the implementation that uses YARN (in Hadoop 2 and later).

> It's important to realize that the old and new MapReduce APIs are not the same thing as the MapReduce 1 and MapReduce 2 implementations. The APIs are user-facing client-side features and determine how you write MapReduce programs (see Appendix D), whereas the implementations are just different ways of running MapReduce programs. All four combinations are supported: both the old and new MapReduce APIs run on both MapReduce 1 and 2.

In MapReduce 1, there are two types of daemon that control the job execution process: a *jobtracker* and one or more *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.

In MapReduce 1, the jobtracker takes care of both job scheduling (matching tasks with tasktrackers) and task progress monitoring (keeping track of tasks, restarting failed or slow tasks, and doing task bookkeeping, such as maintaining counter totals). By contrast, in YARN these responsibilities are handled by separate entities: the resource manager and an application master (one for each MapReduce job). The jobtracker is also responsible for storing job history for completed jobs, although it is possible to run a job history server as a separate daemon to take the load off the jobtracker. In YARN, the equivalent role is the timeline server, which stores application history.[5]

The YARN equivalent of a tasktracker is a node manager. The mapping is summarized in Table 4-1.

Table 4-1. A comparison of MapReduce 1 and YARN components

MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

YARN was designed to address many of the limitations in MapReduce 1. The benefits to using YARN include the following:

Scalability
: YARN can run on larger clusters than MapReduce 1. MapReduce 1 hits scalability bottlenecks in the region of 4,000 nodes and 40,000 tasks,[6] stemming from the fact that the jobtracker has to manage both jobs *and* tasks. YARN overcomes these limitations by virtue of its split resource manager/application master architecture: it is designed to scale up to 10,000 nodes and 100,000 tasks.

  In contrast to the jobtracker, each instance of an application—here, a MapReduce job—has a dedicated application master, which runs for the duration of the application. This model is actually closer to the original Google MapReduce paper, which describes how a master process is started to coordinate map and reduce tasks running on a set of workers.

Availability
: High availability (HA) is usually achieved by replicating the state needed for another daemon to take over the work needed to provide the service, in the event of the service daemon failing. However, the large amount of rapidly changing complex state in the jobtracker's memory (each task status is updated every few seconds, for example) makes it very difficult to retrofit HA into the jobtracker service.

  With the jobtracker's responsibilities split between the resource manager and application master in YARN, making the service highly available became a divide-and-conquer problem: provide HA for the resource manager, then for YARN applications (on a per-application basis). And indeed, Hadoop 2 supports HA both

5. As of Hadoop 2.5.1, the YARN timeline server does not yet store MapReduce job history, so a MapReduce job history server daemon is still needed (see "Cluster Setup and Installation" on page 288).

6. Arun C. Murthy, "The Next Generation of Apache Hadoop MapReduce," February 14, 2011.

for the resource manager and for the application master for MapReduce jobs. Failure recovery in YARN is discussed in more detail in "Failures" on page 193.

Utilization
: In MapReduce 1, each tasktracker is configured with a static allocation of fixed-size "slots," which are divided into map slots and reduce slots at configuration time. A map slot can only be used to run a map task, and a reduce slot can only be used for a reduce task.

  In YARN, a node manager manages a pool of resources, rather than a fixed number of designated slots. MapReduce running on YARN will not hit the situation where a reduce task has to wait because only map slots are available on the cluster, which can happen in MapReduce 1. If the resources to run the task are available, then the application will be eligible for them.

  Furthermore, resources in YARN are fine grained, so an application can make a request for what it needs, rather than for an indivisible slot, which may be too big (which is wasteful of resources) or too small (which may cause a failure) for the particular task.

Multitenancy
: In some ways, the biggest benefit of YARN is that it opens up Hadoop to other types of distributed application beyond MapReduce. MapReduce is just one YARN application among many.

  It is even possible for users to run different versions of MapReduce on the same YARN cluster, which makes the process of upgrading MapReduce more manageable. (Note, however, that some parts of MapReduce, such as the job history server and the shuffle handler, as well as YARN itself, still need to be upgraded across the cluster.)

Since Hadoop 2 is widely used and is the latest stable version, in the rest of this book the term "MapReduce" refers to MapReduce 2 unless otherwise stated. Chapter 7 looks in detail at how MapReduce running on YARN works.

Scheduling in YARN

In an ideal world, the requests that a YARN application makes would be granted immediately. In the real world, however, resources are limited, and on a busy cluster, an application will often need to wait to have some of its requests fulfilled. It is the job of the YARN scheduler to allocate resources to applications according to some defined policy. Scheduling in general is a difficult problem and there is no one "best" policy, which is why YARN provides a choice of schedulers and configurable policies. We look at these next.

Scheduler Options

Three schedulers are available in YARN: the FIFO, Capacity, and Fair Schedulers. The FIFO Scheduler places applications in a queue and runs them in the order of submission (first in, first out). Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served, and so on.

The FIFO Scheduler has the merit of being simple to understand and not needing any configuration, but it's not suitable for shared clusters. Large applications will use all the resources in a cluster, so each application has to wait its turn. On a shared cluster it is better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow long-running jobs to complete in a timely manner, while still allowing users who are running concurrent smaller ad hoc queries to get results back in a reasonable time.

The difference between schedulers is illustrated in Figure 4-3, which shows that under the FIFO Scheduler (i) the small job is blocked until the large job completes.

With the Capacity Scheduler (ii in Figure 4-3), a separate dedicated queue allows the small job to start as soon as it is submitted, although this is at the cost of overall cluster utilization since the queue capacity is reserved for jobs in that queue. This means that the large job finishes later than when using the FIFO Scheduler.

With the Fair Scheduler (iii in Figure 4-3), there is no need to reserve a set amount of capacity, since it will dynamically balance resources between all running jobs. Just after the first (large) job starts, it is the only job running, so it gets all the resources in the cluster. When the second (small) job starts, it is allocated half of the cluster resources so that each job is using its fair share of resources.

Note that there is a lag between the time the second job starts and when it receives its fair share, since it has to wait for resources to free up as containers used by the first job complete. After the small job completes and no longer requires resources, the large job goes back to using the full cluster capacity again. The overall effect is both high cluster utilization and timely small job completion.

Figure 4-3 contrasts the basic operation of the three schedulers. In the next two sections, we examine some of the more advanced configuration options for the Capacity and Fair Schedulers.

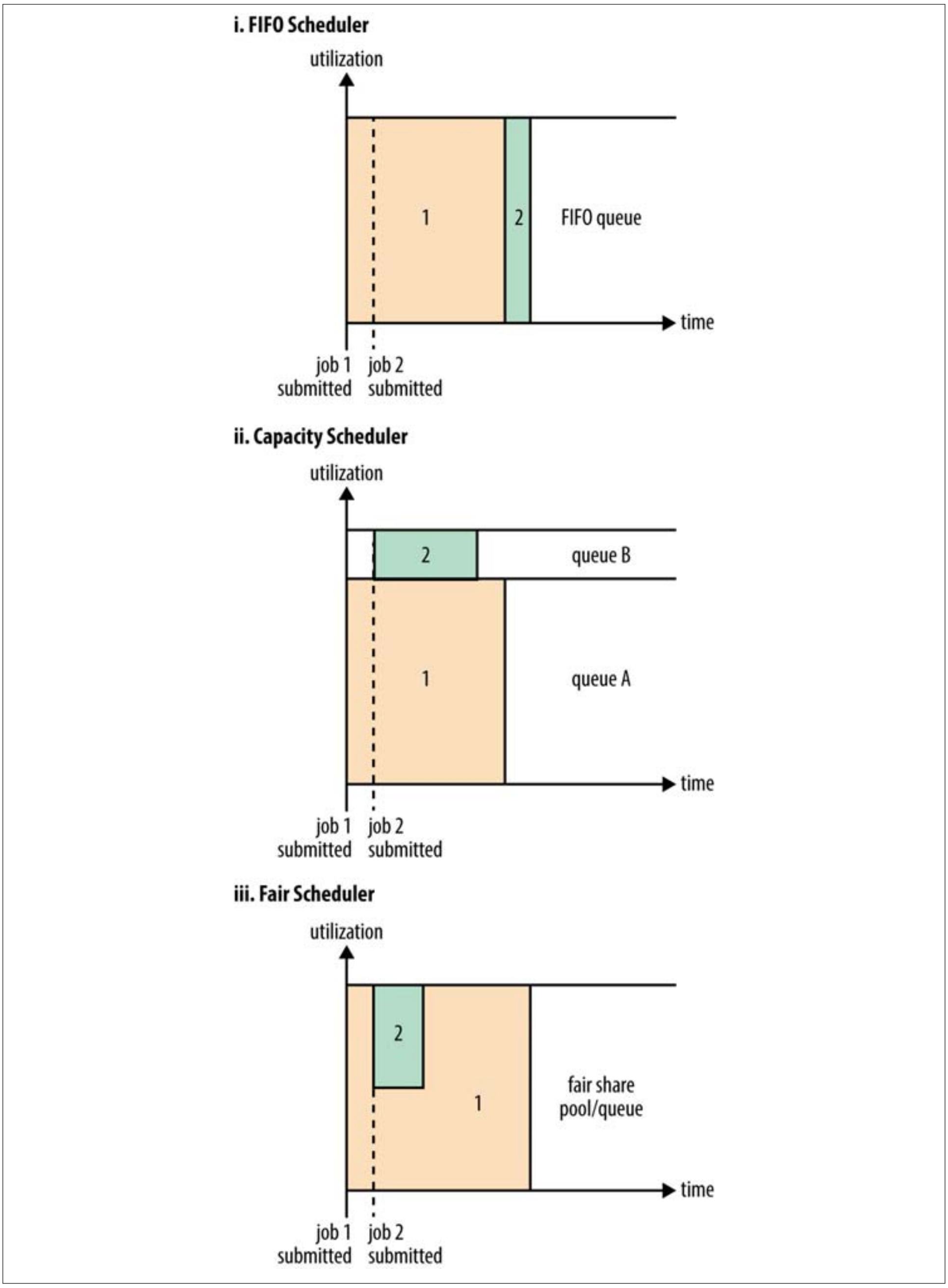

Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

CHAPTER 7

How MapReduce Works

In this chapter, we look at how MapReduce in Hadoop works in detail. This knowledge provides a good foundation for writing more advanced MapReduce programs, which we will cover in the following two chapters.

Anatomy of a MapReduce Job Run

You can run a MapReduce job with a single method call: `submit()` on a `Job` object (you can also call `waitForCompletion()`, which submits the job if it hasn't been submitted already, then waits for it to finish).[1] This method call 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 7-1. At the highest level, there are five independent entities:[2]

- The client, which submits the MapReduce job.
- The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
- The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
- The MapReduce application master, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers.

1. In the old MapReduce API, you can call `JobClient.submitJob(conf)` or `JobClient.runJob(conf)`.
2. Not discussed in this section are the job history server daemon (for retaining job history data) and the shuffle handler auxiliary service (for serving map outputs to reduce tasks).

- The distributed filesystem (normally HDFS, covered in Chapter 3), which is used for sharing job files between the other entities.

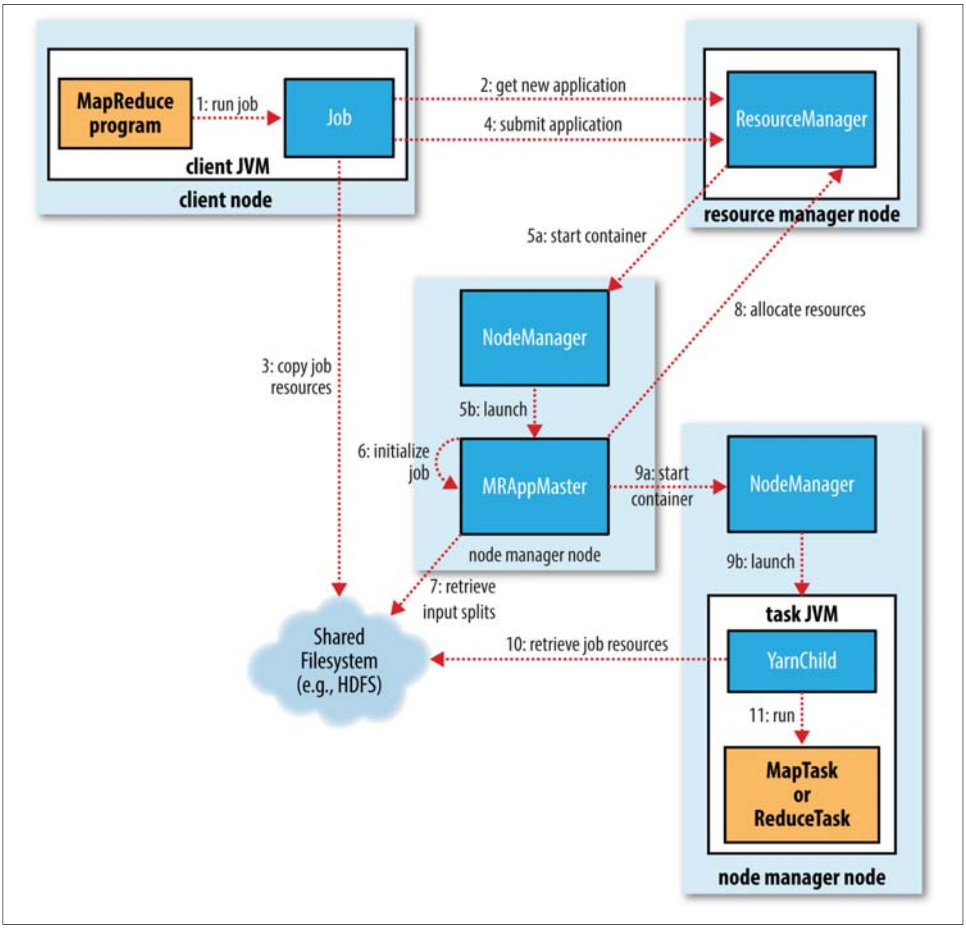

Figure 7-1. How Hadoop runs a MapReduce job

Job Submission

The `submit()` method on `Job` creates an internal `JobSubmitter` instance and calls `submitJobInternal()` on it (step 1 in Figure 7-1). Having submitted the job, `waitFor Completion()` polls the job's progress once per second and reports the progress to the console if it has changed since the last report. When the job completes successfully, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by `JobSubmitter` does the following:

- Asks the resource manager for a new application ID, used for the MapReduce job ID (step 2).
- 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), 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 shared filesystem in a directory named after the job ID (step 3). The job JAR is copied with a high replication factor (controlled by the `mapreduce.client.submit.file.replication` property, which defaults to 10) so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
- Submits the job by calling `submitApplication()` on the resource manager (step 4).

Job Initialization

When the resource manager receives a call to its `submitApplication()` method, it hands off the request to the YARN scheduler. The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management (steps 5a and 5b).

The application master for MapReduce jobs is a Java application whose main class is `MRAppMaster`. It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks (step 6). Next, it retrieves the input splits computed in the client from the shared filesystem (step 7). It then creates a map task object for each split, as well as a number of reduce task objects determined by the `mapreduce.job.reduces` property (set by the `setNumReduceTasks()` method on `Job`). Tasks are given IDs at this point.

The application master must decide how to run the tasks that make up the MapReduce job. If the job is small, the application master may choose to run the tasks in the same JVM as itself. This happens when it judges that the overhead of allocating and running tasks in new containers outweighs the gain to be had in running them in parallel, compared to running them sequentially on one node. Such a job is said to be *uberized*, or run as an *uber task*.

What qualifies as a small job? By default, a small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block. (Note that these values may be changed for a job by setting `mapreduce.job.ubertask.maxmaps`, `mapreduce.job.ubertask.maxreduces`, and `mapreduce.job.ubertask.maxbytes`.) Uber tasks must be enabled explicitly (for an individual job, or across the cluster) by setting `mapreduce.job.ubertask.enable` to `true`.

Finally, before any tasks can be run, the application master calls the `setupJob()` method on the `OutputCommitter`. For `FileOutputCommitter`, which is the default, it will create the final output directory for the job and the temporary working space for the task output. The commit protocol is described in more detail in "Output Committers" on page 206.

Task Assignment

If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8). Requests for map tasks are made first and with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start (see "Shuffle and Sort" on page 197). Requests for reduce tasks are not made until 5% of map tasks have completed (see "Reduce slow start" on page 308).

Reduce tasks can run anywhere in the cluster, but requests for map tasks have data locality constraints that the scheduler tries to honor (see "Resource Requests" on page 81). In the optimal case, the task is *data local*—that is, running on the same node that the split resides on. Alternatively, the task may be *rack local*: on the same rack, but not the same node, as the split. Some tasks are neither data local nor rack local and retrieve their data from a different rack than the one they are running on. For a particular job run, you can determine the number of tasks that ran at each locality level by looking at the job's counters (see Table 9-6).

Requests also specify memory requirements and CPUs for tasks. By default, each map and reduce task is allocated 1,024 MB of memory and one virtual core. The values are configurable on a per-job basis (subject to minimum and maximum values described in "Memory settings in YARN and MapReduce" on page 301) via the following properties: `mapreduce.map.memory.mb`, `mapreduce.reduce.memory.mb`, `mapreduce.map.cpu.vcores` and `mapreduce.reduce.cpu.vcores`.

Task Execution

Once a task has been assigned resources for a container on a particular node by the resource manager's scheduler, the application master starts the container by contacting the node manager (steps 9a and 9b). The task is executed by a Java application whose main class is `YarnChild`. Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache (step 10; see "Distributed Cache" on page 274). Finally, it runs the map or reduce task (step 11).

The `YarnChild` runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions (or even in `YarnChild`) don't affect the node manager—by causing it to crash or hang, for example.

Each task can perform setup and commit actions, which are run in the same JVM as the task itself and are determined by the `OutputCommitter` for the job (see "Output Committers" on page 206). For file-based jobs, the commit action moves the task output from a temporary location to its final location. The commit protocol ensures that when speculative execution is enabled (see "Speculative Execution" on page 204), only one of the duplicate tasks is committed and the other is aborted.

Streaming

Streaming runs special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it (Figure 7-2).

The Streaming task communicates with the process (which may be written in any language) using standard input and output streams. During execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs back to the Java process. From the node manager's point of view, it is as if the child process ran the map or reduce code itself.

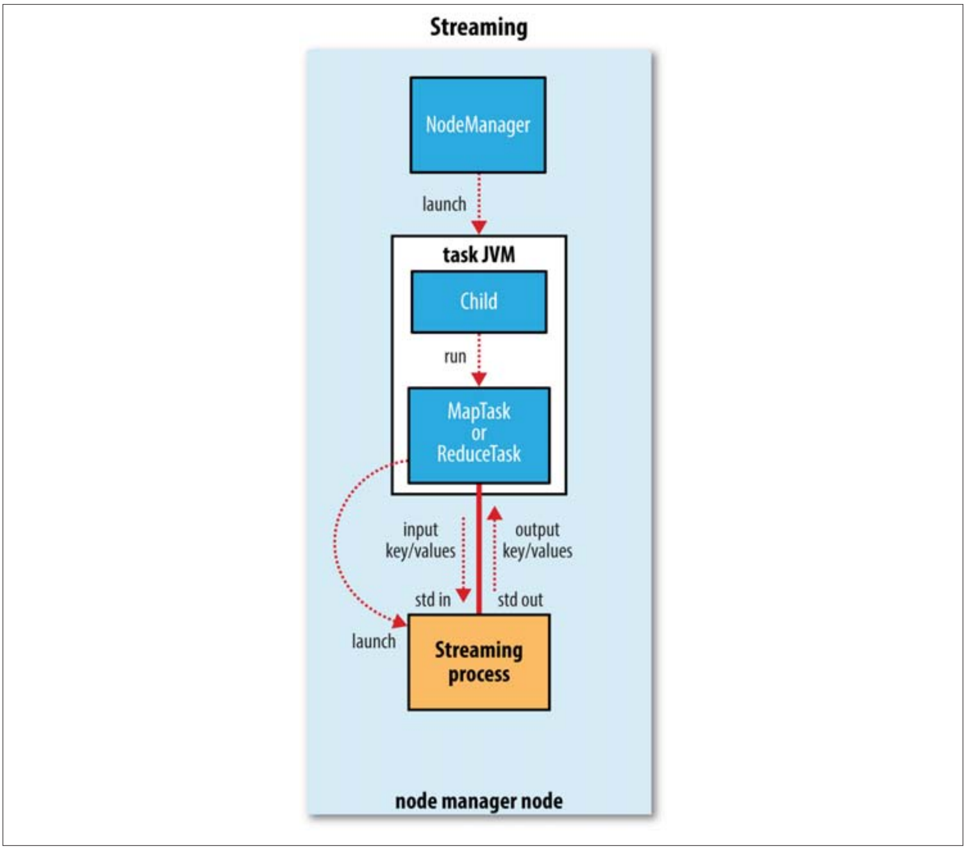

Figure 7-2. The relationship of the Streaming executable to the node manager and the task container

Progress and Status Updates

MapReduce jobs are long-running batch jobs, taking anything from tens of seconds to hours to run. Because this can be a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a *status*, which includes such things as the state of the job or task (e.g., running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code). These statuses change over the course of the job, so how do they get communicated back to the client?

When a task is running, it keeps track of its *progress* (i.e., the proportion of the task completed). For map tasks, this is the proportion of the input that has been processed. For reduce tasks, it's a little more complex, but the system can still estimate the proportion of the reduce input processed. It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle (see "Shuffle and Sort" on page 197). For example, if the task has run the reducer on half its input, the task's progress is 5/6, since it has completed the copy and sort phases (1/3 each) and is halfway through the reduce phase (1/6).

What Constitutes Progress in MapReduce?

Progress is not always measurable, but nevertheless, it tells Hadoop that a task is doing something. For example, a task writing output records is making progress, even when it cannot be expressed as a percentage of the total number that will be written (because the latter figure may not be known, even by the task producing the output).

Progress reporting is important, as Hadoop will not fail a task that's making progress. All of the following operations constitute progress:

- Reading an input record (in a mapper or reducer)
- Writing an output record (in a mapper or reducer)
- Setting the status description (via `Reporter`'s or `TaskAttemptContext`'s `setStatus()` method)
- Incrementing a counter (using `Reporter`'s `incrCounter()` method or `Counter`'s `increment()` method)
- Calling `Reporter`'s or `TaskAttemptContext`'s `progress()` method

Tasks also have a set of counters that count various events as the task runs (we saw an example in "A test run" on page 27), which are either built into the framework, such as the number of map output records written, or defined by users.

As the map or reduce task runs, the child process communicates with its parent application master through the *umbilical* interface. The task reports its progress and status (including counters) back to its application master, which has an aggregate view of the job, every three seconds over the umbilical interface.

The resource manager web UI displays all the running applications with links to the web UIs of their respective application masters, each of which displays further details on the MapReduce job, including its progress.

During the course of the job, the client receives the latest status by polling the application master every second (the interval is set via `mapreduce.client.progressmonitor.pollinterval`). Clients can also use `Job`'s `getStatus()` method to obtain a `JobStatus` instance, which contains all of the status information for the job.

The process is illustrated in Figure 7-3.

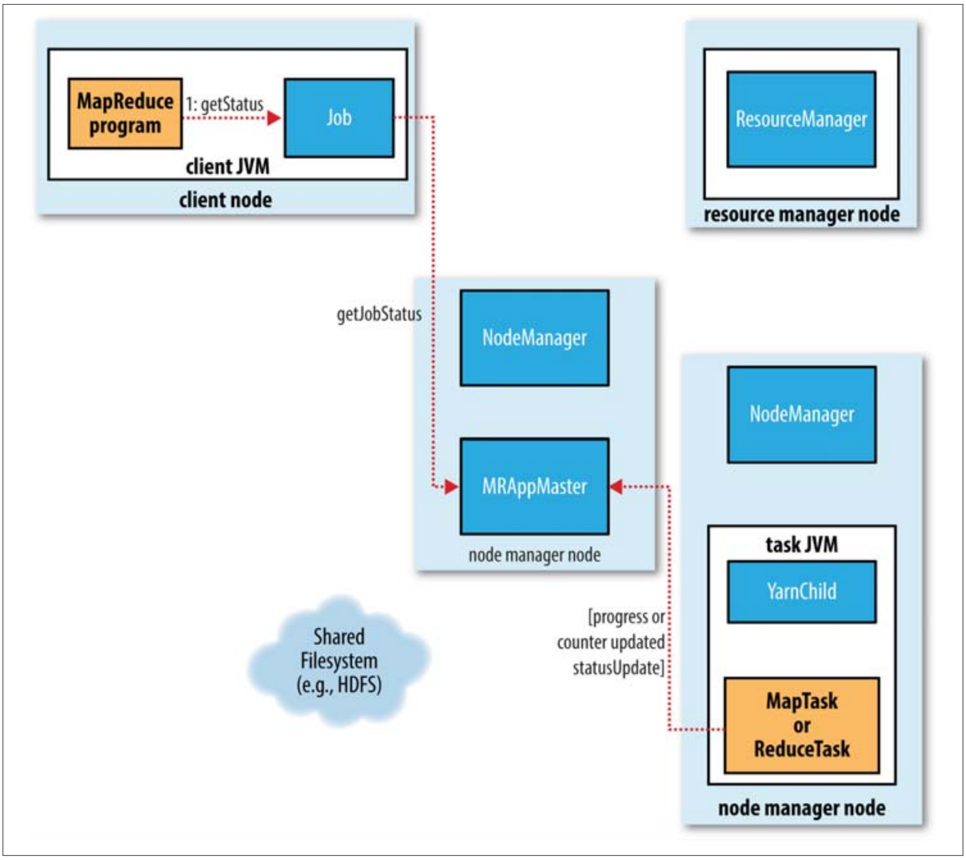

Figure 7-3. How status updates are propagated through the MapReduce system

Job Completion

When the application master receives a notification that the last task for a job is complete, it changes the status for the job to "successful." Then, when the `Job` polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the `waitForCompletion()` method. Job statistics and counters are printed to the console at this point.

The application master also sends an HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the `mapreduce.job.end-notification.url` property.

Finally, on job completion, the application master and the task containers clean up their working state (so intermediate output is deleted), and the `OutputCommitter`'s `commitJob()` method is called. Job information is archived by the job history server to enable later interrogation by users if desired.

Failures

In the real world, user code is buggy, processes crash, and machines fail. One of the major benefits of using Hadoop is its ability to handle such failures and allow your job to complete successfully. We need to consider the failure of any of the following entities: the task, the application master, the node manager, and the resource manager.

Task Failure

Consider first the case of the task failing. The most common occurrence of this failure is when user code in the map or reduce task throws a runtime exception. If this happens, the task JVM reports the error back to its parent application master before it exits. The error ultimately makes it into the user logs. The application master marks the task attempt as *failed*, and frees up the container so its resources are available for another task.

For Streaming tasks, if the Streaming process exits with a nonzero exit code, it is marked as failed. This behavior is governed by the `stream.non.zero.exit.is.failure` property (the default is `true`).

Another failure mode is the sudden exit of the task JVM—perhaps there is a JVM bug that causes the JVM to exit for a particular set of circumstances exposed by the MapReduce user code. In this case, the node manager notices that the process has exited and informs the application master so it can mark the attempt as failed.

Hanging tasks are dealt with differently. The application master notices that it hasn't received a progress update for a while and proceeds to mark the task as failed. The task JVM process will be killed automatically after this period.[3] The timeout period after which tasks are considered failed is normally 10 minutes and can be configured on a per-job basis (or a cluster basis) by setting the `mapreduce.task.timeout` property to a value in milliseconds.

Setting the timeout to a value of zero disables the timeout, so long-running tasks are never marked as failed. In this case, a hanging task will never free up its container, and over time there may be cluster slowdown as a result. This approach should therefore be avoided, and making sure that a task is reporting progress periodically should suffice (see "What Constitutes Progress in MapReduce?" on page 191).

3. If a Streaming process hangs, the node manager will kill it (along with the JVM that launched it) only in the following circumstances: either `yarn.nodemanager.container-executor.class` is set to `org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor`, or the default container executor is being used and the `setsid` command is available on the system (so that the task JVM and any processes it launches are in the same process group). In any other case, orphaned Streaming processes will accumulate on the system, which will impact utilization over time.

When the application master is notified of a task attempt that has failed, it will reschedule execution of the task. The application master will try to avoid rescheduling the task on a node manager where it has previously failed. Furthermore, if a task fails four times, it will not be retried again. This value is configurable. The maximum number of attempts to run a task is controlled by the `mapreduce.map.maxattempts` property for map tasks and `mapreduce.reduce.maxattempts` for reduce tasks. By default, if any task fails four times (or whatever the maximum number of attempts is configured to), the whole job fails.

For some applications, it is undesirable to abort the job if a few tasks fail, as it may be possible to use the results of the job despite some failures. In this case, the maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job. Map tasks and reduce tasks are controlled independently, using the `mapreduce.map.failures.maxpercent` and `mapreduce.reduce.failures.maxpercent` properties.

A task attempt may also be *killed*, which is different from it failing. A task attempt may be killed because it is a speculative duplicate (for more information on this topic, see "Speculative Execution" on page 204), or because the node manager it was running on failed and the application master marked all the task attempts running on it as killed. Killed task attempts do not count against the number of attempts to run the task (as set by `mapreduce.map.maxattempts` and `mapreduce.reduce.maxattempts`), because it wasn't the task's fault that an attempt was killed.

Users may also kill or fail task attempts using the web UI or the command line (type `mapred job` to see the options). Jobs may be killed by the same mechanisms.

Application Master Failure

Just like MapReduce tasks are given several attempts to succeed (in the face of hardware or network failures), applications in YARN are retried in the event of failure. The maximum number of attempts to run a MapReduce application master is controlled by the `mapreduce.am.max-attempts` property. The default value is 2, so if a MapReduce application master fails twice it will not be tried again and the job will fail.

YARN imposes a limit for the maximum number of attempts for any YARN application master running on the cluster, and individual applications may not exceed this limit. The limit is set by `yarn.resourcemanager.am.max-attempts` and defaults to 2, so if you want to increase the number of MapReduce application master attempts, you will have to increase the YARN setting on the cluster, too.

The way recovery works is as follows. An application master sends periodic heartbeats to the resource manager, and in the event of application master failure, the resource manager will detect the failure and start a new instance of the master running in a new container (managed by a node manager). In the case of the MapReduce application master, it will use the job history to recover the state of the tasks that were already run by the (failed) application so they don't have to be rerun. Recovery is enabled by default, but can be disabled by setting `yarn.app.mapreduce.am.job.recovery.enable` to `false`.

The MapReduce client polls the application master for progress reports, but if its application master fails, the client needs to locate the new instance. During job initialization, the client asks the resource manager for the application master's address, and then caches it so it doesn't overload the resource manager with a request every time it needs to poll the application master. If the application master fails, however, the client will experience a timeout when it issues a status update, at which point the client will go back to the resource manager to ask for the new application master's address. This process is transparent to the user.

Node Manager Failure

If a node manager fails by crashing or running very slowly, it will stop sending heartbeats to the resource manager (or send them very infrequently). The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes (this is configured, in milliseconds, via the `yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms` property) and remove it from its pool of nodes to schedule containers on.

Any task or application master running on the failed node manager will be recovered using the mechanisms described in the previous two sections. In addition, the application master arranges for map tasks that were run and completed successfully on the failed node manager to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed node manager's local filesystem may not be accessible to the reduce task.

Node managers may be *blacklisted* if the number of failures for the application is high, even if the node manager itself has not failed. Blacklisting is done by the application master, and for MapReduce the application master will try to reschedule tasks on different nodes if more than three tasks fail on a node manager. The user may set the threshold with the `mapreduce.job.maxtaskfailures.per.tracker` job property.

> Note that the resource manager does not do blacklisting across applications (at the time of writing), so tasks from new jobs may be scheduled on bad nodes even if they have been blacklisted by an application master running an earlier job.

Resource Manager Failure

Failure of the resource manager is serious, because without it, neither jobs nor task containers can be launched. In the default configuration, the resource manager is a single point of failure, since in the (unlikely) event of machine failure, all running jobs fail—and can't be recovered.

To achieve high availability (HA), it is necessary to run a pair of resource managers in an active-standby configuration. If the active resource manager fails, then the standby can take over without a significant interruption to the client.

Information about all the running applications is stored in a highly available state store (backed by ZooKeeper or HDFS), so that the standby can recover the core state of the failed active resource manager. Node manager information is not stored in the state store since it can be reconstructed relatively quickly by the new resource manager as the node managers send their first heartbeats. (Note also that tasks are not part of the resource manager's state, since they are managed by the application master. Thus, the amount of state to be stored is therefore much more manageable than that of the jobtracker in MapReduce 1.)

When the new resource manager starts, it reads the application information from the state store, then restarts the application masters for all the applications running on the cluster. This does not count as a failed application attempt (so it does not count against `yarn.resourcemanager.am.max-attempts`), since the application did not fail due to an error in the application code, but was forcibly killed by the system. In practice, the application master restart is not an issue for MapReduce applications since they recover the work done by completed tasks (as we saw in "Application Master Failure" on page 194).

The transition of a resource manager from standby to active is handled by a failover controller. The default failover controller is an automatic one, which uses ZooKeeper leader election to ensure that there is only a single active resource manager at one time. Unlike in HDFS HA (see "HDFS High Availability" on page 48), the failover controller does not have to be a standalone process, and is embedded in the resource manager by default for ease of configuration. It is also possible to configure manual failover, but this is not recommended.

Clients and node managers must be configured to handle resource manager failover, since there are now two possible resource managers to communicate with. They try connecting to each resource manager in a round-robin fashion until they find the active one. If the active fails, then they will retry until the standby becomes active.

Shuffle and Sort

MapReduce makes the guarantee that the input to every reducer is sorted by key. The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the *shuffle*.[4] In this section, we look at how the shuffle works, as a basic understanding will be helpful should you need to optimize a MapReduce program. The shuffle is an area of the codebase where refinements and improvements are continually being made, so the following description necessarily conceals many details. In many ways, the shuffle is the heart of MapReduce and is where the "magic" happens.

The Map Side

When the map function starts producing output, it is not simply written to disk. The process is more involved, and takes advantage of buffering writes in memory and doing some presorting for efficiency reasons. Figure 7-4 shows what happens.

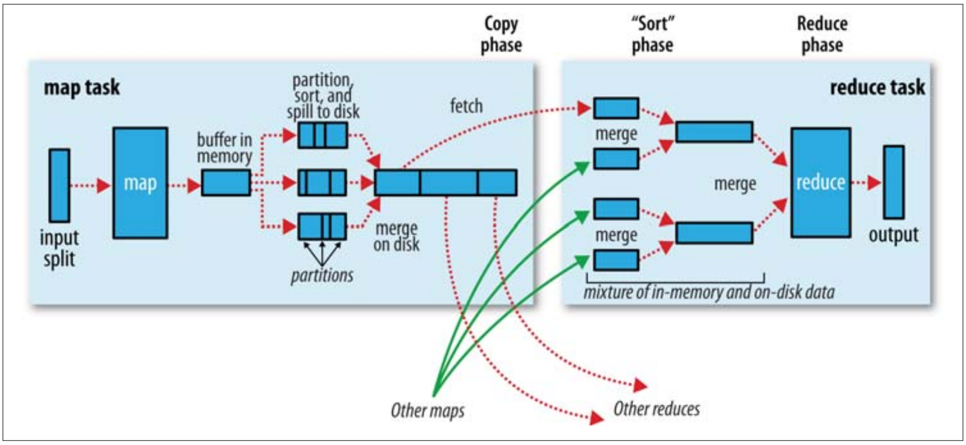

Figure 7-4. Shuffle and sort in MapReduce

Each map task has a circular memory buffer that it writes the output to. The buffer is 100 MB by default (the size can be tuned by changing the `mapreduce.task.io.sort.mb` property). When the contents of the buffer reach a certain threshold size (`mapreduce.map.sort.spill.percent`, which has the default value 0.80, or 80%), a background thread will start to *spill* the contents to disk. Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time,

4. The term *shuffle* is actually imprecise, since in some contexts it refers to only the part of the process where map outputs are fetched by reduce tasks. In this section, we take it to mean the whole process, from the point where a map produces output to where a reduce consumes input.

the map will block until the spill is complete. Spills are written in round-robin fashion to the directories specified by the `mapreduce.cluster.local.dir` property, in a job-specific subdirectory.

Before it writes to disk, the thread first divides the data into partitions corresponding to the reducers that they will ultimately be sent to. Within each partition, the background thread performs an in-memory sort by key, and if there is a combiner function, it is run on the output of the sort. Running the combiner function makes for a more compact map output, so there is less data to write to local disk and to transfer to the reducer.

Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written its last output record, there could be several spill files. Before the task is finished, the spill files are merged into a single partitioned and sorted output file. The configuration property `mapreduce.task.io.sort.factor` controls the maximum number of streams to merge at once; the default is 10.

If there are at least three spill files (set by the `mapreduce.map.combine.minspills` property), the combiner is run again before the output file is written. Recall that combiners may be run repeatedly over the input without affecting the final result. If there are only one or two spills, the potential reduction in map output size is not worth the overhead in invoking the combiner, so it is not run again for this map output.

It is often a good idea to compress the map output as it is written to disk, because doing so makes it faster to write to disk, saves disk space, and reduces the amount of data to transfer to the reducer. By default, the output is not compressed, but it is easy to enable this by setting `mapreduce.map.output.compress` to `true`. The compression library to use is specified by `mapreduce.map.output.compress.codec`; see "Compression" on page 100 for more on compression formats.

The output file's partitions are made available to the reducers over HTTP. The maximum number of worker threads used to serve the file partitions is controlled by the `mapreduce.shuffle.max.threads` property; this setting is per node manager, not per map task. The default of 0 sets the maximum number of threads to twice the number of processors on the machine.

The Reduce Side

Let's turn now to the reduce part of the process. The map output file is sitting on the local disk of the machine that ran the map task (note that although map outputs always get written to local disk, reduce outputs may not be), but now it is needed by the machine that is about to run the reduce task for the partition. Moreover, the reduce task needs the map output for its particular partition from several map tasks across the cluster. The map tasks may finish at different times, so the reduce task starts copying their outputs as soon as each completes. This is known as the *copy phase* of the reduce task. The reduce task has a small number of copier threads so that it can fetch map outputs in parallel.

The default is five threads, but this number can be changed by setting the `mapreduce.reduce.shuffle.parallelcopies` property.

> How do reducers know which machines to fetch map output from?
>
> As map tasks complete successfully, they notify their application master using the heartbeat mechanism. Therefore, for a given job, the application master knows the mapping between map outputs and hosts. A thread in the reducer periodically asks the master for map output hosts until it has retrieved them all.
>
> Hosts do not delete map outputs from disk as soon as the first reducer has retrieved them, as the reducer may subsequently fail. Instead, they wait until they are told to delete them by the application master, which is after the job has completed.

Map outputs are copied to the reduce task JVM's memory if they are small enough (the buffer's size is controlled by `mapreduce.reduce.shuffle.input.buffer.percent`, which specifies the proportion of the heap to use for this purpose); otherwise, they are copied to disk. When the in-memory buffer reaches a threshold size (controlled by `mapreduce.reduce.shuffle.merge.percent`) or reaches a threshold number of map outputs (`mapreduce.reduce.merge.inmem.threshold`), it is merged and spilled to disk. If a combiner is specified, it will be run during the merge to reduce the amount of data written to disk.

As the copies accumulate on disk, a background thread merges them into larger, sorted files. This saves some time merging later on. Note that any map outputs that were compressed (by the map task) have to be decompressed in memory in order to perform a merge on them.

When all the map outputs have been copied, the reduce task moves into the *sort phase* (which should properly be called the *merge* phase, as the sorting was carried out on the map side), which merges the map outputs, maintaining their sort ordering. This is done in rounds. For example, if there were 50 map outputs and the *merge factor* was 10 (the default, controlled by the `mapreduce.task.io.sort.factor` property, just like in the map's merge), there would be five rounds. Each round would merge 10 files into 1, so at the end there would be 5 intermediate files.

Rather than have a final round that merges these five files into a single sorted file, the merge saves a trip to disk by directly feeding the reduce function in what is the last phase: the *reduce phase*. This final merge can come from a mixture of in-memory and on-disk segments.

The number of files merged in each round is actually more subtle than this example suggests. The goal is to merge the minimum number of files to get to the merge factor for the final round. So if there were 40 files, the merge would not merge 10 files in each of the four rounds to get 4 files. Instead, the first round would merge only 4 files, and the subsequent three rounds would merge the full 10 files. The 4 merged files and the 6 (as yet unmerged) files make a total of 10 files for the final round. The process is illustrated in Figure 7-5.

Note that this does not change the number of rounds; it's just an optimization to minimize the amount of data that is written to disk, since the final round always merges directly into the reduce.

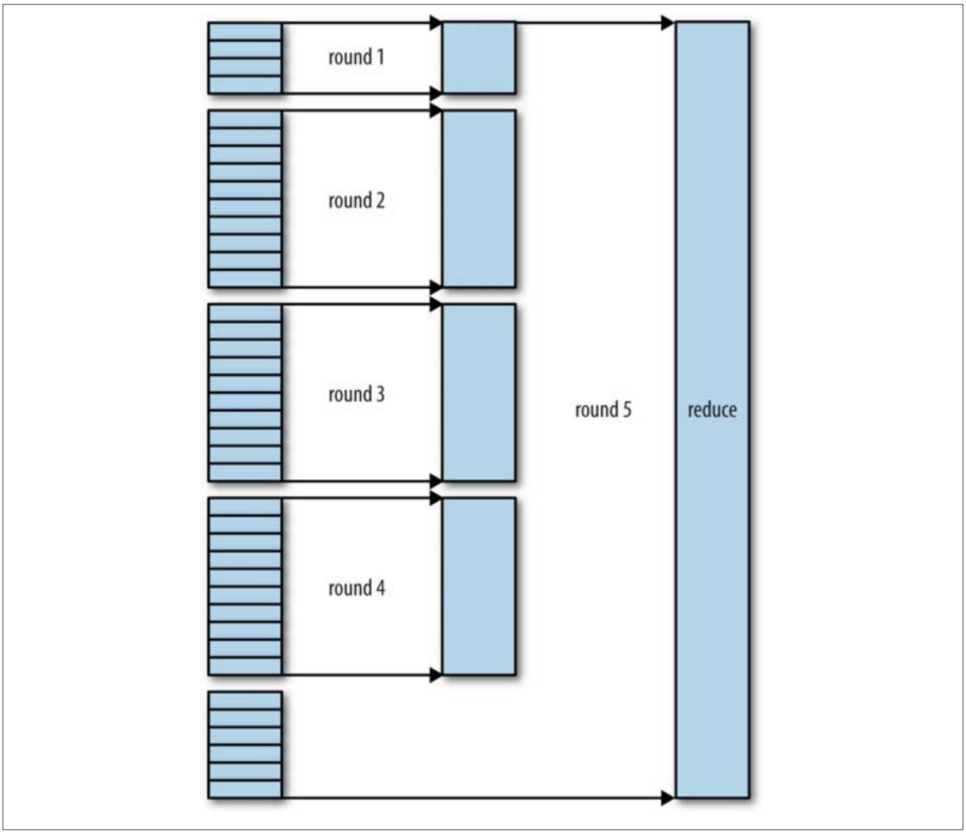

Figure 7-5. Efficiently merging 40 file segments with a merge factor of 10

During the reduce phase, the reduce function is invoked for each key in the sorted output. The output of this phase is written directly to the output filesystem, typically

HDFS. In the case of HDFS, because the node manager is also running a datanode, the first block replica will be written to the local disk.

Configuration Tuning

We are now in a better position to understand how to tune the shuffle to improve MapReduce performance. The relevant settings, which can be used on a per-job basis (except where noted), are summarized in Tables 7-1 and 7-2, along with the defaults, which are good for general-purpose jobs.

The general principle is to give the shuffle as much memory as possible. However, there is a trade-off, in that you need to make sure that your map and reduce functions get enough memory to operate. This is why it is best to write your map and reduce functions to use as little memory as possible—certainly they should not use an unbounded amount of memory (avoid accumulating values in a map, for example).

The amount of memory given to the JVMs in which the map and reduce tasks run is set by the `mapred.child.java.opts` property. You should try to make this as large as possible for the amount of memory on your task nodes; the discussion in "Memory settings in YARN and MapReduce" on page 301 goes through the constraints to consider.

On the map side, the best performance can be obtained by avoiding multiple spills to disk; one is optimal. If you can estimate the size of your map outputs, you can set the `mapreduce.task.io.sort.*` properties appropriately to minimize the number of spills. In particular, you should increase `mapreduce.task.io.sort.mb` if you can. There is a MapReduce counter (`SPILLED_RECORDS`; see "Counters" on page 247) that counts the total number of records that were spilled to disk over the course of a job, which can be useful for tuning. Note that the counter includes both map- and reduce-side spills.

On the reduce side, the best performance is obtained when the intermediate data can reside entirely in memory. This does not happen by default, since for the general case all the memory is reserved for the reduce function. But if your reduce function has light memory requirements, setting `mapreduce.reduce.merge.inmem.threshold` to 0 and `mapreduce.reduce.input.buffer.percent` to 1.0 (or a lower value; see Table 7-2) may bring a performance boost.

In April 2008, Hadoop won the general-purpose terabyte sort benchmark (as discussed in "A Brief History of Apache Hadoop" on page 12), and one of the optimizations used was keeping the intermediate data in memory on the reduce side.

More generally, Hadoop uses a buffer size of 4 KB by default, which is low, so you should increase this across the cluster (by setting `io.file.buffer.size`; see also "Other Hadoop Properties" on page 307).

Example 9-8. Reduce function for secondary sort in Python

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

import sys

last_group = None
for line in sys.stdin:
  val = line.strip()
  (year, temp) = val.split("\t")
  group = year
  if last_group != group:
    print val
    last_group = group
```

When we run the Streaming program, we get the same output as the Java version.

Finally, note that `KeyFieldBasedPartitioner` and `KeyFieldBasedComparator` are not confined to use in Streaming programs; they are applicable to Java MapReduce programs, too.

Joins

MapReduce can perform joins between large datasets, but writing the code to do joins from scratch is fairly involved. Rather than writing MapReduce programs, you might consider using a higher-level framework such as Pig, Hive, Cascading, Cruc, or Spark, in which join operations are a core part of the implementation.

Let's briefly consider the problem we are trying to solve. We have two datasets—for example, the weather stations database and the weather records—and we want to reconcile the two. Let's say we want to see each station's history, with the station's metadata inlined in each output row. This is illustrated in Figure 9-2.

How we implement the join depends on how large the datasets are and how they are partitioned. If one dataset is large (the weather records) but the other one is small enough to be distributed to each node in the cluster (as the station metadata is), the join can be effected by a MapReduce job that brings the records for each station together (a partial sort on station ID, for example). The mapper or reducer uses the smaller dataset to look up the station metadata for a station ID, so it can be written out with each record. See "Side Data Distribution" on page 273 for a discussion of this approach, where we focus on the mechanics of distributing the data to nodes in the cluster.

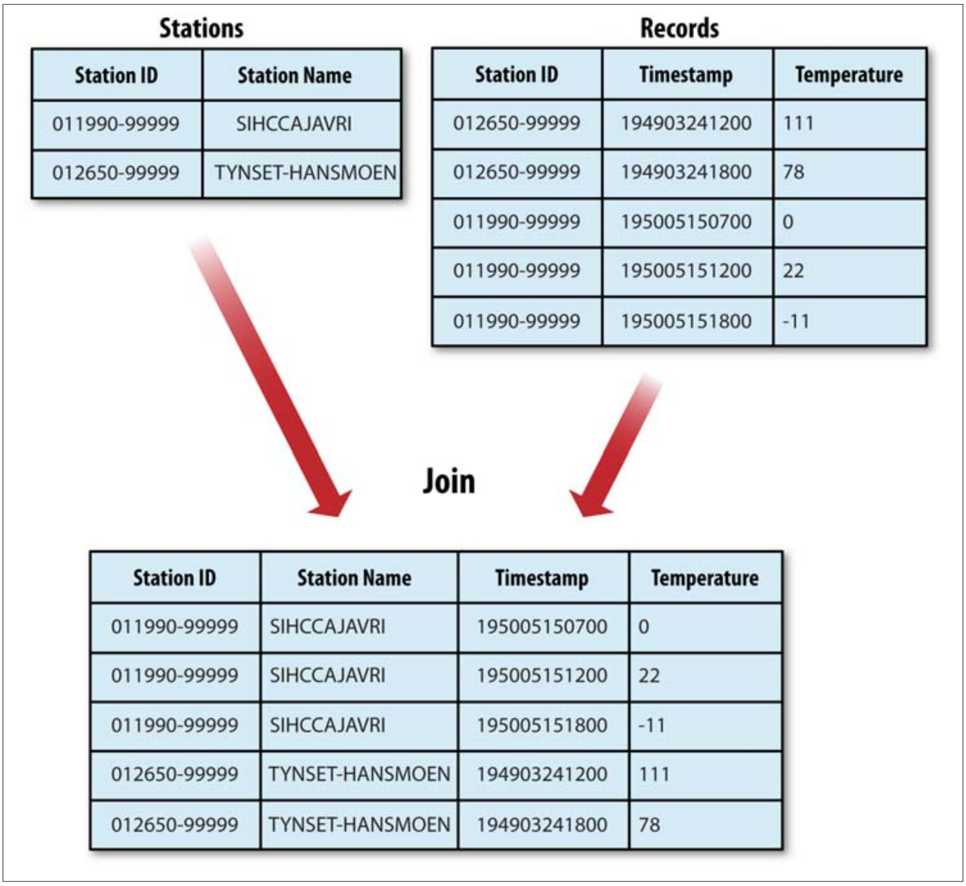

Figure 9-2. Inner join of two datasets

If the join is performed by the mapper it is called a *map-side join*, whereas if it is performed by the reducer it is called a *reduce-side join*.

If both datasets are too large for either to be copied to each node in the cluster, we can still join them using MapReduce with a map-side or reduce-side join, depending on how the data is structured. One common example of this case is a user database and a log of some user activity (such as access logs). For a popular service, it is not feasible to distribute the user database (or the logs) to all the MapReduce nodes.

Map-Side Joins

A map-side join between large inputs works by performing the join before the data reaches the map function. For this to work, though, the inputs to each map must be partitioned and sorted in a particular way. Each input dataset must be divided into the same number of partitions, and it must be sorted by the same key (the join key) in each source. All the records for a particular key must reside in the same partition. This may sound like a strict requirement (and it is), but it actually fits the description of the output of a MapReduce job.

A map-side join can be used to join the outputs of several jobs that had the same number of reducers, the same keys, and output files that are not splittable (by virtue of being smaller than an HDFS block or being gzip compressed, for example). In the context of the weather example, if we ran a partial sort on the stations file by station ID, and another identical sort on the records, again by station ID and with the same number of reducers, then the two outputs would satisfy the conditions for running a map-side join.

You use a `CompositeInputFormat` from the `org.apache.hadoop.mapreduce.join` package to run a map-side join. The input sources and join type (inner or outer) for `CompositeInputFormat` are configured through a join expression that is written according to a simple grammar. The package documentation has details and examples.

The `org.apache.hadoop.examples.Join` example is a general-purpose command-line program for running a map-side join, since it allows you to run a MapReduce job for any specified mapper and reducer over multiple inputs that are joined with a given join operation.

Reduce-Side Joins

A reduce-side join is more general than a map-side join, in that the input datasets don't have to be structured in any particular way, but it is less efficient because both datasets have to go through the MapReduce shuffle. The basic idea is that the mapper tags each record with its source and uses the join key as the map output key, so that the records with the same key are brought together in the reducer. We use several ingredients to make this work in practice:

Multiple inputs
: The input sources for the datasets generally have different formats, so it is very convenient to use the `MultipleInputs` class (see "Multiple Inputs" on page 237) to separate the logic for parsing and tagging each source.

Secondary sort
: As described, the reducer will see the records from both sources that have the same key, but they are not guaranteed to be in any particular order. However, to perform the join, it is important to have the data from one source before that from the other. For the weather data join, the station record must be the first of the values seen for each key, so the reducer can fill in the weather records with the station name and emit them straightaway. Of course, it would be possible to receive the records in any order if we buffered them in memory, but this should be avoided because the

number of records in any group may be very large and exceed the amount of memory available to the reducer.

We saw in "Secondary Sort" on page 262 how to impose an order on the values for each key that the reducers see, so we use this technique here.

To tag each record, we use `TextPair` (discussed in Chapter 5) for the keys (to store the station ID) and the tag. The only requirement for the tag values is that they sort in such a way that the station records come before the weather records. This can be achieved by tagging station records as `0` and weather records as `1`. The mapper classes to do this are shown in Examples 9-9 and 9-10.

Example 9-9. Mapper for tagging station records for a reduce-side join

```
public class JoinStationMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcStationMetadataParser parser = new NcdcStationMetadataParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    if (parser.parse(value)) {
      context.write(new TextPair(parser.getStationId(), "0"),
          new Text(parser.getStationName()));
    }
  }
}
```

Example 9-10. Mapper for tagging weather records for a reduce-side join

```
public class JoinRecordMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    parser.parse(value);
    context.write(new TextPair(parser.getStationId(), "1"), value);
  }

}
```

The reducer knows that it will receive the station record first, so it extracts its name from the value and writes it out as a part of every output record (Example 9-11).

Example 9-11. Reducer for joining tagged station records with tagged weather records

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {

  @Override
  protected void reduce(TextPair key, Iterable<Text> values, Context context)
```

```
      throws IOException, InterruptedException {
    Iterator<Text> iter = values.iterator();
    Text stationName = new Text(iter.next());
    while (iter.hasNext()) {
      Text record = iter.next();
      Text outValue = new Text(stationName.toString() + "\t" + record.toString());
      context.write(key.getFirst(), outValue);
    }
  }
}
```

The code assumes that every station ID in the weather records has exactly one matching record in the station dataset. If this were not the case, we would need to generalize the code to put the tag into the value objects, by using another `TextPair`. The `reduce()` method would then be able to tell which entries were station names and detect (and handle) missing or duplicate entries before processing the weather records.

> Because objects in the reducer's values iterator are reused (for efficiency purposes), it is vital that the code makes a copy of the first `Text` object from the `values` iterator:
>
> ```
> Text stationName = new Text(iter.next());
> ```
>
> If the copy is not made, the `stationName` reference will refer to the value just read when it is turned into a string, which is a bug.

Tying the job together is the driver class, shown in Example 9-12. The essential point here is that we partition and group on the first part of the key, the station ID, which we do with a custom `Partitioner` (`KeyPartitioner`) and a custom group comparator, `FirstComparator` (from `TextPair`).

Example 9-12. Application to join weather records with station names

```
public class JoinRecordWithStationName extends Configured implements Tool {

  public static class KeyPartitioner extends Partitioner<TextPair, Text> {
    @Override
    public int getPartition(TextPair key, Text value, int numPartitions) {
      return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 3) {
      JobBuilder.printUsage(this, "<ncdc input> <station input> <output>");
      return -1;
    }

    Job job = new Job(getConf(), "Join weather records with station names");
```

```
    job.setJarByClass(getClass());

    Path ncdcInputPath = new Path(args[0]);
    Path stationInputPath = new Path(args[1]);
    Path outputPath = new Path(args[2]);

    MultipleInputs.addInputPath(job, ncdcInputPath,
        TextInputFormat.class, JoinRecordMapper.class);
    MultipleInputs.addInputPath(job, stationInputPath,
        TextInputFormat.class, JoinStationMapper.class);
    FileOutputFormat.setOutputPath(job, outputPath);

    job.setPartitionerClass(KeyPartitioner.class);
    job.setGroupingComparatorClass(TextPair.FirstComparator.class);

    job.setMapOutputKeyClass(TextPair.class);

    job.setReducerClass(JoinReducer.class);

    job.setOutputKeyClass(Text.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new JoinRecordWithStationName(), args);
    System.exit(exitCode);
  }
}
```

Running the program on the sample data yields the following output:

```
011990-99999    SIHCCAJAVRI                 0067011990999991950051507004...
011990-99999    SIHCCAJAVRI                 0043011990999991950051512004...
011990-99999    SIHCCAJAVRI                 0043011990999991950051518004...
012650-99999    TYNSET-HANSMOEN             0043012650999991949032412004...
012650-99999    TYNSET-HANSMOEN             0043012650999991949032418004...
```

Side Data Distribution

Side data can be defined as extra read-only data needed by a job to process the main dataset. The challenge is to make side data available to all the map or reduce tasks (which are spread across the cluster) in a convenient and efficient fashion.

Using the Job Configuration

You can set arbitrary key-value pairs in the job configuration using the various setter methods on `Configuration` (or `JobConf` in the old MapReduce API). This is very useful when you need to pass a small piece of metadata to your tasks.