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.

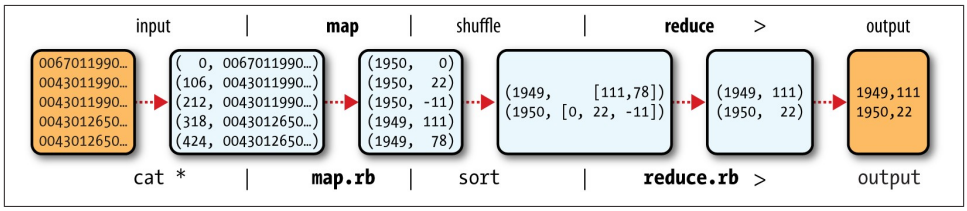

Figure 2-1. MapReduce logical data flow

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

```
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.

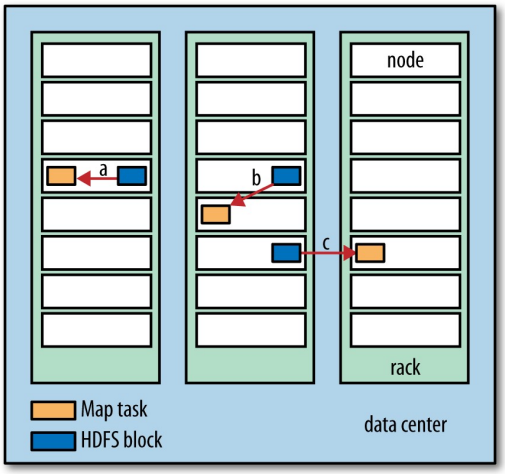

Figure 2-2. Data-local (a), rack-local (b), and off-rack (c) map tasks

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.

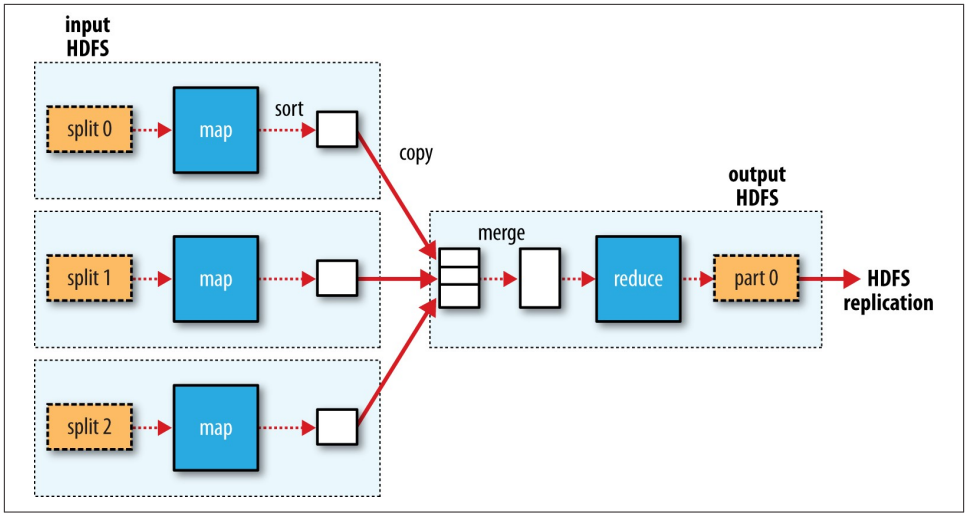

Figure 2-3. MapReduce data flow with a single reduce task

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.

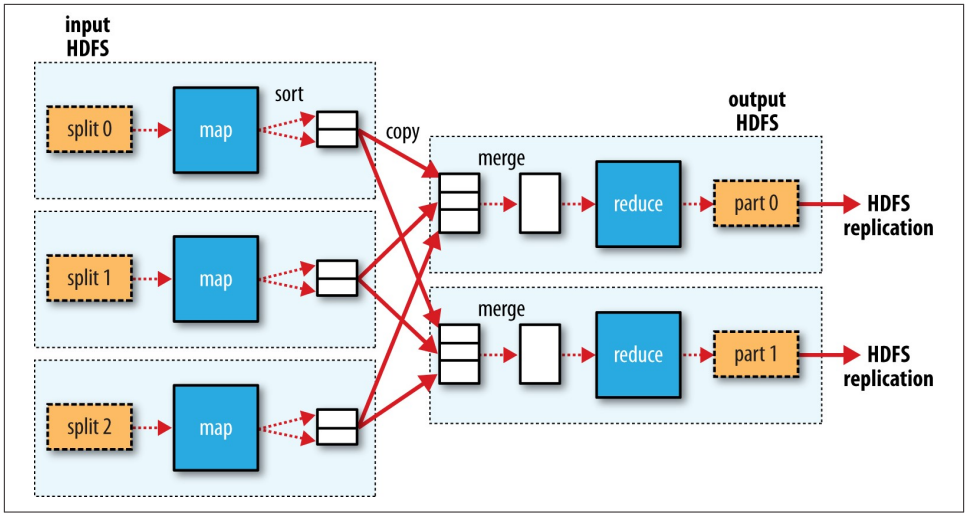

Figure 2-4. MapReduce data flow with multiple reduce tasks

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.

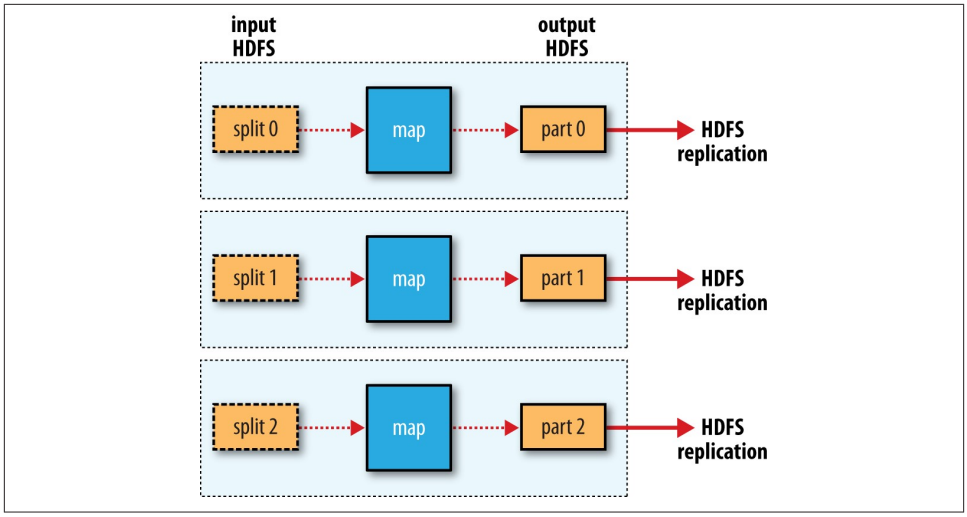

Figure 2-5. MapReduce data flow with no reduce tasks

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.

Example 2-7. Map function for maximum temperature in Ruby

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

STDIN.each_line do |line|
  val = line
  year, temp, q = val[15,4], val[87,5], val[92,1]
  puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)
end
```

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

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

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

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

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

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

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

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

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

```
  if last_key && last_key != key
    puts "#{last_key}\t#{max_val}"
    last_key, max_val = key, val.to_i
  else
    last_key, max_val = key, [max_val, val.to_i].max
  end
end
puts "#{last_key}\t#{max_val}" if last_key
```

Again, the program iterates over lines from standard input, but this time we have to store some state as we process each key group. In this case, the keys are the years, and we store the last key seen and the maximum temperature seen so far for that key. The MapReduce framework ensures that the keys are ordered, so we know that if a key is different from the previous one, we have moved into a new key group. In contrast to the Java API, where you are provided an iterator over each key group, in Streaming you have to find key group boundaries in your program.

For each line, we pull out the key and value. Then, if we've just finished a group (`last_key && last_key != key`), we write the key and the maximum temperature for that group, separated by a tab character, before resetting the maximum temperature for the new key. If we haven't just finished a group, we just update the maximum temperature for the current key.

The last line of the program ensures that a line is written for the last key group in the input.

We can now simulate the whole MapReduce pipeline with a Unix pipeline (which is equivalent to the Unix pipeline shown in Figure 2-1):

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

The output is the same as that of the Java program, so the next step is to run it using Hadoop itself.

The `hadoop` command doesn't support a Streaming option; instead, you specify the Streaming JAR file along with the `jar` option. Options to the Streaming program specify the input and output paths and the map and reduce scripts. This is what it looks like:

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

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

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

Note also the use of `-files`, which we use when running Streaming programs on the cluster to ship the scripts to the cluster.

Python

Streaming supports any programming language that can read from standard input and write to standard output, so for readers more familiar with Python, here's the same example again.[5] The map script is in Example 2-9, and the reduce script is in Example 2-10.

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

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

import re
import sys

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

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

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

import sys

(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
  (key, val) = line.strip().split("\t")
  if last_key and last_key != key:
    print "%s\t%s" % (last_key, max_val)
    (last_key, max_val) = (key, int(val))
  else:
    (last_key, max_val) = (key, max(max_val, int(val)))
```

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

```
if last_key:
  print "%s\t%s" % (last_key, max_val)
```

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

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

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).

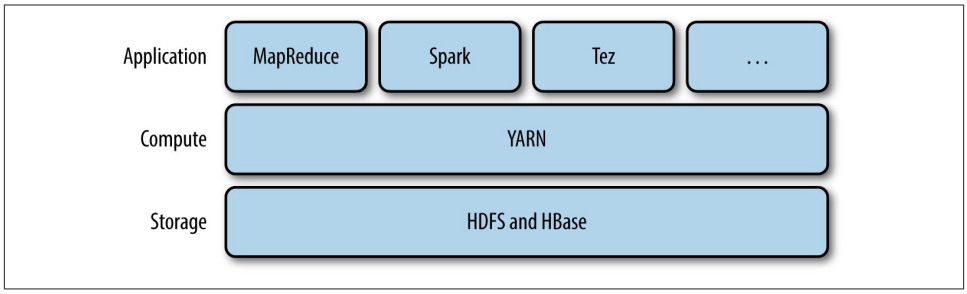

Figure 4-1. YARN applications

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.

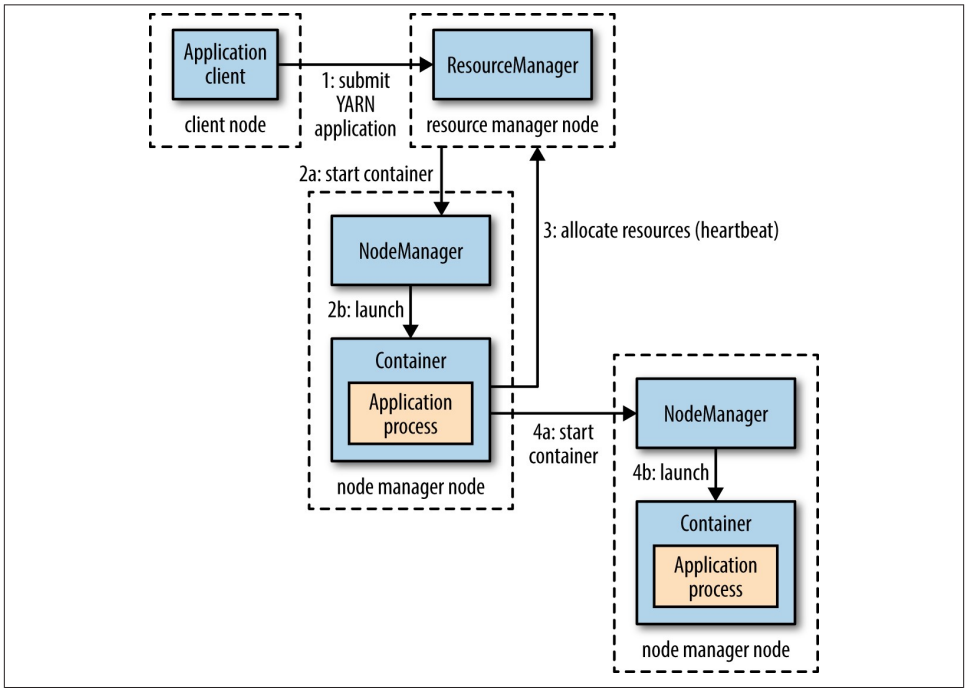

Figure 4-2. How YARN runs an application

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.

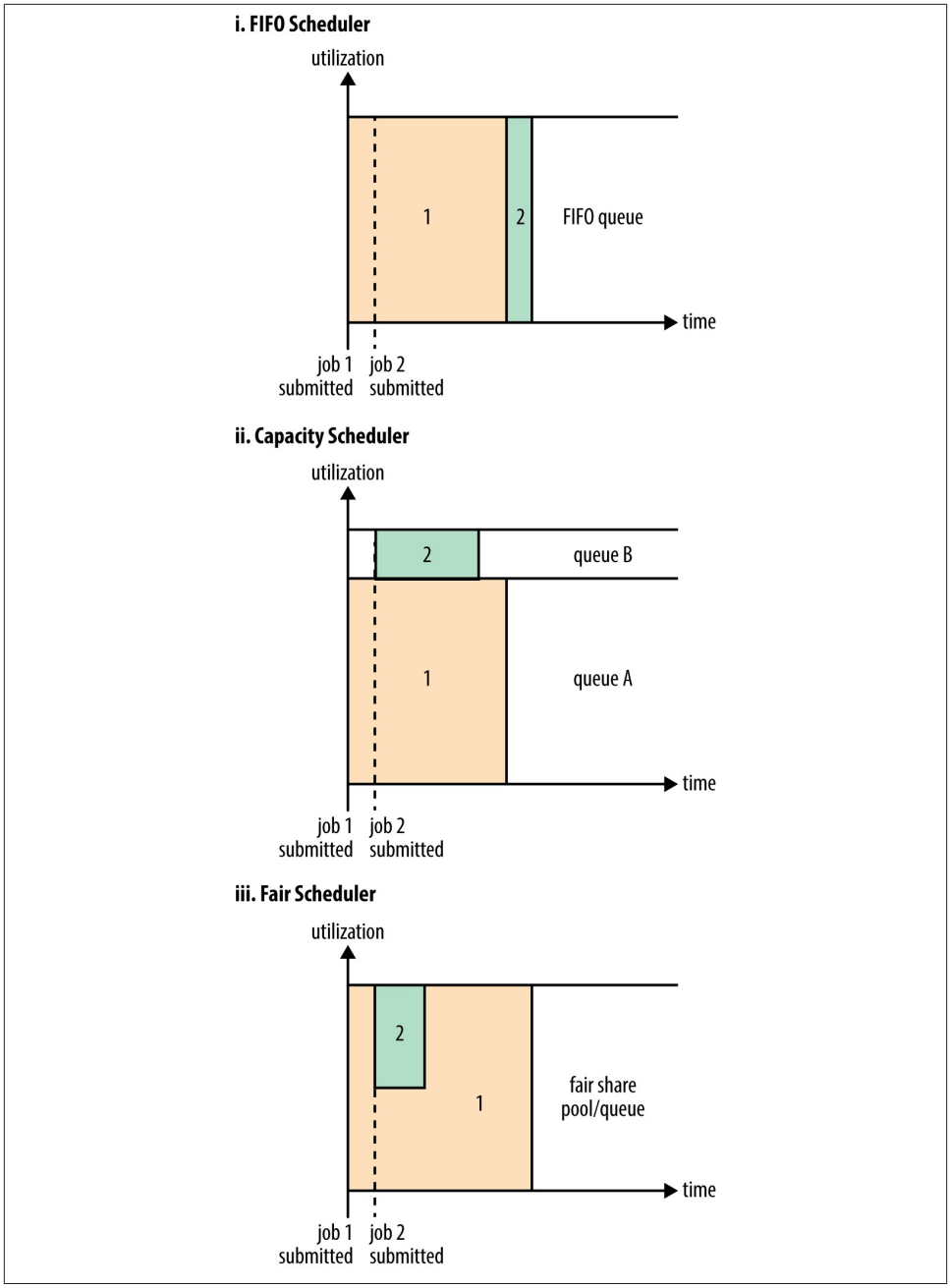

Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

Capacity Scheduler Configuration

The Capacity Scheduler allows sharing of a Hadoop cluster along organizational lines, whereby each organization is allocated a certain capacity of the overall cluster. Each organization is set up with a dedicated queue that is configured to use a given fraction of the cluster capacity. Queues may be further divided in hierarchical fashion, allowing each organization to share its cluster allowance between different groups of users within the organization. Within a queue, applications are scheduled using FIFO scheduling.

As we saw in Figure 4-3, a single job does not use more resources than its queue's capacity. However, if there is more than one job in the queue and there are idle resources available, then the Capacity Scheduler may allocate the spare resources to jobs in the queue, even if that causes the queue's capacity to be exceeded.[7] This behavior is known as *queue elasticity*.

In normal operation, the Capacity Scheduler does not preempt containers by forcibly killing them,[8] so if a queue is under capacity due to lack of demand, and then demand increases, the queue will only return to capacity as resources are released from other queues as containers complete. It is possible to mitigate this by configuring queues with a maximum capacity so that they don't eat into other queues' capacities too much. This is at the cost of queue elasticity, of course, so a reasonable trade-off should be found by trial and error.

Imagine a queue hierarchy that looks like this:

```
root
├── prod
└── dev
    ├── eng
    └── science
```

The listing in Example 4-1 shows a sample Capacity Scheduler configuration file, called *capacity-scheduler.xml*, for this hierarchy. It defines two queues under the `root` queue, `prod` and `dev`, which have 40% and 60% of the capacity, respectively. Notice that a particular queue is configured by setting configuration properties of the form `yarn.scheduler.capacity.<queue-path>.<sub-property>`, where `<queue-path>` is the hierarchical (dotted) path of the queue, such as `root.prod`.

7. If the property `yarn.scheduler.capacity.<queue-path>.user-limit-factor` is set to a value larger than 1 (the default), then a single job is allowed to use more than its queue's capacity.

8. However, the Capacity Scheduler can perform work-preserving preemption, where the resource manager asks applications to return containers to balance capacity.

Example 4-1. A basic configuration file for the Capacity Scheduler

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.queues</name>
    <value>eng,science</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>40</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>60</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.maximum-capacity</name>
    <value>75</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.eng.capacity</name>
    <value>50</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.science.capacity</name>
    <value>50</value>
  </property>
</configuration>
```

As you can see, the `dev` queue is further divided into `eng` and `science` queues of equal capacity. So that the `dev` queue does not use up all the cluster resources when the `prod` queue is idle, it has its maximum capacity set to 75%. In other words, the `prod` queue always has 25% of the cluster available for immediate use. Since no maximum capacities have been set for other queues, it's possible for jobs in the `eng` or `science` queues to use all of the `dev` queue's capacity (up to 75% of the cluster), or indeed for the `prod` queue to use the entire cluster.

Beyond configuring queue hierarchies and capacities, there are settings to control the maximum number of resources a single user or application can be allocated, how many applications can be running at any one time, and ACLs on queues. See the reference page for details.

Queue placement

The way that you specify which queue an application is placed in is specific to the application. For example, in MapReduce, you set the property `mapreduce.job.queuename` to the name of the queue you want to use. If the queue does not exist, then you'll get an error at submission time. If no queue is specified, applications will be placed in a queue called `default`.

For the Capacity Scheduler, the queue name should be the last part of the hierarchical name since the full hierarchical name is not recognized. So, for the preceding example configuration, `prod` and `eng` are OK, but `root.dev.eng` and `dev.eng` do not work.

Fair Scheduler Configuration

The Fair Scheduler attempts to allocate resources so that all running applications get the same share of resources. Figure 4-3 showed how fair sharing works for applications in the same queue; however, fair sharing actually works *between* queues, too, as we'll see next.

The terms *queue* and *pool* are used interchangeably in the context of the Fair Scheduler.

To understand how resources are shared between queues, imagine two users *A* and *B*, each with their own queue (Figure 4-4). *A* starts a job, and it is allocated all the resources available since there is no demand from *B*. Then *B* starts a job while *A*'s job is still running, and after a while each job is using half of the resources, in the way we saw earlier. Now if *B* starts a second job while the other jobs are still running, it will share its resources with *B*'s other job, so each of *B*'s jobs will have one-fourth of the resources, while *A*'s will continue to have half. The result is that resources are shared fairly between users.

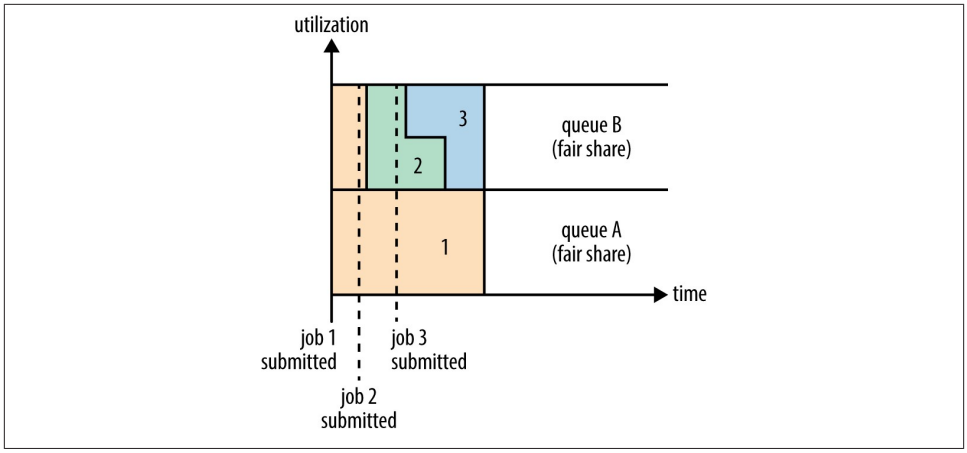

Figure 4-4. Fair sharing between user queues

Enabling the Fair Scheduler

The scheduler in use is determined by the setting of `yarn.resourcemanager.scheduler.class`. The Capacity Scheduler is used by default (although the Fair Scheduler is the default in some Hadoop distributions, such as CDH), but this can be changed by setting `yarn.resourcemanager.scheduler.class` in *yarn-site.xml* to the fully qualified classname of the scheduler, `org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler`.

Queue configuration

The Fair Scheduler is configured using an allocation file named *fair-scheduler.xml* that is loaded from the classpath. (The name can be changed by setting the property `yarn.scheduler.fair.allocation.file`.) In the absence of an allocation file, the Fair Scheduler operates as described earlier: each application is placed in a queue named after the user and queues are created dynamically when users submit their first applications.

Per-queue configuration is specified in the allocation file. This allows configuration of hierarchical queues like those supported by the Capacity Scheduler. For example, we can define `prod` and `dev` queues like we did for the Capacity Scheduler using the allocation file in Example 4-2.

Example 4-2. An allocation file for the Fair Scheduler

```
<?xml version="1.0"?>
<allocations>
  <defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>

  <queue name="prod">
```

```
    <weight>40</weight>
    <schedulingPolicy>fifo</schedulingPolicy>
  </queue>

  <queue name="dev">
    <weight>60</weight>
    <queue name="eng" />
    <queue name="science" />
  </queue>

  <queuePlacementPolicy>
    <rule name="specified" create="false" />
    <rule name="primaryGroup" create="false" />
    <rule name="default" queue="dev.eng" />
  </queuePlacementPolicy>
</allocations>
```

The queue hierarchy is defined using nested `queue` elements. All queues are children of the `root` queue, even if not actually nested in a `root` `queue` element. Here we subdivide the `dev` queue into a queue called `eng` and another called `science`.

Queues can have weights, which are used in the fair share calculation. In this example, the cluster allocation is considered fair when it is divided into a 40:60 proportion between `prod` and `dev`. The `eng` and `science` queues do not have weights specified, so they are divided evenly. Weights are not quite the same as percentages, even though the example uses numbers that add up to 100 for the sake of simplicity. We could have specified weights of 2 and 3 for the `prod` and `dev` queues to achieve the same queue weighting.

When setting weights, remember to consider the default queue and dynamically created queues (such as queues named after users). These are not specified in the allocation file, but still have weight 1.

Queues can have different scheduling policies. The default policy for queues can be set in the top-level `defaultQueueSchedulingPolicy` element; if it is omitted, fair scheduling is used. Despite its name, the Fair Scheduler also supports a FIFO (`fifo`) policy on queues, as well as Dominant Resource Fairness (`drf`), described later in the chapter.

The policy for a particular queue can be overridden using the `schedulingPolicy` element for that queue. In this case, the `prod` queue uses FIFO scheduling since we want each production job to run serially and complete in the shortest possible amount of time. Note that fair sharing is still used to divide resources between the `prod` and `dev` queues, as well as between (and within) the `eng` and `science` queues.

Although not shown in this allocation file, queues can be configured with minimum and maximum resources, and a maximum number of running applications. (See the reference page for details.) The minimum resources setting is not a hard limit, but rather is used by the scheduler to prioritize resource allocations. If two queues are below their fair share, then the one that is furthest below its minimum is allocated resources first. The minimum resource setting is also used for preemption, discussed momentarily.

Queue placement

The Fair Scheduler uses a rules-based system to determine which queue an application is placed in. In Example 4-2, the `queuePlacementPolicy` element contains a list of rules, each of which is tried in turn until a match occurs. The first rule, `specified`, places an application in the queue it specified; if none is specified, or if the specified queue doesn't exist, then the rule doesn't match and the next rule is tried. The `primaryGroup` rule tries to place an application in a queue with the name of the user's primary Unix group; if there is no such queue, rather than creating it, the next rule is tried. The `default` rule is a catch-all and always places the application in the `dev.eng` queue.

The `queuePlacementPolicy` can be omitted entirely, in which case the default behavior is as if it had been specified with the following:

```
<queuePlacementPolicy>
  <rule name="specified" />
  <rule name="user" />
</queuePlacementPolicy>
```

In other words, unless the queue is explicitly specified, the user's name is used for the queue, creating it if necessary.

Another simple queue placement policy is one where all applications are placed in the same (default) queue. This allows resources to be shared fairly between applications, rather than users. The definition is equivalent to this:

```
<queuePlacementPolicy>
  <rule name="default" />
</queuePlacementPolicy>
```

It's also possible to set this policy without using an allocation file, by setting `yarn.scheduler.fair.user-as-default-queue` to `false` so that applications will be placed in the default queue rather than a per-user queue. In addition, `yarn.scheduler.fair.allow-undeclared-pools` should be set to `false` so that users can't create queues on the fly.

Preemption

When a job is submitted to an empty queue on a busy cluster, the job cannot start until resources free up from jobs that are already running on the cluster. To make the time taken for a job to start more predictable, the Fair Scheduler supports *preemption*.

Preemption allows the scheduler to kill containers for queues that are running with more than their fair share of resources so that the resources can be allocated to a queue that is under its fair share. Note that preemption reduces overall cluster efficiency, since the terminated containers need to be reexecuted.

Preemption is enabled globally by setting `yarn.scheduler.fair.preemption` to `true`. There are two relevant preemption timeout settings: one for minimum share and one for fair share, both specified in seconds. By default, the timeouts are not set, so you need to set at least one to allow containers to be preempted.

If a queue waits for as long as its *minimum share preemption timeout* without receiving its minimum guaranteed share, then the scheduler may preempt other containers. The default timeout is set for all queues via the `defaultMinSharePreemptionTimeout` top-level element in the allocation file, and on a per-queue basis by setting the `minShare PreemptionTimeout` element for a queue.

Likewise, if a queue remains below *half* of its fair share for as long as the *fair share preemption timeout*, then the scheduler may preempt other containers. The default timeout is set for all queues via the `defaultFairSharePreemptionTimeout` top-level element in the allocation file, and on a per-queue basis by setting `fairSharePreemp tionTimeout` on a queue. The threshold may also be changed from its default of 0.5 by setting `defaultFairSharePreemptionThreshold` and `fairSharePreemptionThres hold` (per-queue).

Delay Scheduling

All the YARN schedulers try to honor locality requests. On a busy cluster, if an application requests a particular node, there is a good chance that other containers are running on it at the time of the request. The obvious course of action is to immediately loosen the locality requirement and allocate a container on the same rack. However, it has been observed in practice that waiting a short time (no more than a few seconds) can dramatically increase the chances of being allocated a container on the requested node, and therefore increase the efficiency of the cluster. This feature is called *delay scheduling*, and it is supported by both the Capacity Scheduler and the Fair Scheduler.

Every node manager in a YARN cluster periodically sends a heartbeat request to the resource manager—by default, one per second. Heartbeats carry information about the node manager's running containers and the resources available for new containers, so each heartbeat is a potential *scheduling opportunity* for an application to run a container.

When using delay scheduling, the scheduler doesn't simply use the first scheduling opportunity it receives, but waits for up to a given maximum number of scheduling opportunities to occur before loosening the locality constraint and taking the next scheduling opportunity.

For the Capacity Scheduler, delay scheduling is configured by setting `yarn.scheduler.capacity.node-locality-delay` to a positive integer representing the number of scheduling opportunities that it is prepared to miss before loosening the node constraint to match any node in the same rack.

The Fair Scheduler also uses the number of scheduling opportunities to determine the delay, although it is expressed as a proportion of the cluster size. For example, setting `yarn.scheduler.fair.locality.threshold.node` to 0.5 means that the scheduler should wait until half of the nodes in the cluster have presented scheduling opportunities before accepting another node in the same rack. There is a corresponding property, `yarn.scheduler.fair.locality.threshold.rack`, for setting the threshold before another rack is accepted instead of the one requested.

Dominant Resource Fairness

When there is only a single resource type being scheduled, such as memory, then the concept of capacity or fairness is easy to determine. If two users are running applications, you can measure the amount of memory that each is using to compare the two applications. However, when there are multiple resource types in play, things get more complicated. If one user's application requires lots of CPU but little memory and the other's requires little CPU and lots of memory, how are these two applications compared?

The way that the schedulers in YARN address this problem is to look at each user's dominant resource and use it as a measure of the cluster usage. This approach is called *Dominant Resource Fairness*, or DRF for short.[9] The idea is best illustrated with a simple example.

Imagine a cluster with a total of 100 CPUs and 10 TB of memory. Application *A* requests containers of (2 CPUs, 300 GB), and application *B* requests containers of (6 CPUs, 100 GB). *A*'s request is (2%, 3%) of the cluster, so memory is dominant since its proportion (3%) is larger than CPU's (2%). *B*'s request is (6%, 1%), so CPU is dominant. Since *B*'s container requests are twice as big in the dominant resource (6% versus 3%), it will be allocated half as many containers under fair sharing.

By default DRF is not used, so during resource calculations, only memory is considered and CPU is ignored. The Capacity Scheduler can be configured to use DRF by setting `yarn.scheduler.capacity.resource-calculator` to `org.apache.hadoop.yarn.util.resource.DominantResourceCalculator` in *capacity-scheduler.xml*.

For the Fair Scheduler, DRF can be enabled by setting the top-level element `defaultQueueSchedulingPolicy` in the allocation file to `drf`.

9. DRF was introduced in Ghodsi et al.'s "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types," March 2011.

Further Reading

This chapter has given a short overview of YARN. For more detail, see *Apache Hadoop YARN* by Arun C. Murthy et al. (Addison-Wesley, 2014).

CHAPTER 5
Hadoop I/O

Hadoop comes with a set of primitives for data I/O. Some of these are techniques that are more general than Hadoop, such as data integrity and compression, but deserve special consideration when dealing with multiterabyte datasets. Others are Hadoop tools or APIs that form the building blocks for developing distributed systems, such as serialization frameworks and on-disk data structures.

Data Integrity

Users of Hadoop rightly expect that no data will be lost or corrupted during storage or processing. However, because every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing, when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.

The usual way of detecting corrupted data is by computing a *checksum* for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data. The data is deemed to be corrupt if the newly generated checksum doesn't exactly match the original. This technique doesn't offer any way to fix the data—it is merely error detection. (And this is a reason for not using low-end hardware; in particular, be sure to use ECC memory.) Note that it is possible that it's the checksum that is corrupt, not the data, but this is very unlikely, because the checksum is much smaller than the data.

A commonly used error-detecting code is CRC-32 (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size. CRC-32 is used for checksumming in Hadoop's `ChecksumFileSystem`, while HDFS uses a more efficient variant called CRC-32C.

Data Integrity in HDFS

HDFS transparently checksums all data written to it and by default verifies checksums when reading data. A separate checksum is created for every `dfs.bytes-per-checksum` bytes of data. The default is 512 bytes, and because a CRC-32C checksum is 4 bytes long, the storage overhead is less than 1%.

Datanodes are responsible for verifying the data they receive before storing the data and its checksum. This applies to data that they receive from clients and from other datanodes during replication. A client writing data sends it to a pipeline of datanodes (as explained in Chapter 3), and the last datanode in the pipeline verifies the checksum. If the datanode detects an error, the client receives a subclass of `IOException`, which it should handle in an application-specific manner (for example, by retrying the operation).

When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes. Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified. When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.

In addition to block verification on client reads, each datanode runs a `DataBlockScanner` in a background thread that periodically verifies all the blocks stored on the datanode. This is to guard against corruption due to "bit rot" in the physical storage media. See "Datanode block scanner" on page 328 for details on how to access the scanner reports.

Because HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a `ChecksumException`. The namenode marks the block replica as corrupt so it doesn't direct any more clients to it or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level. Once this has happened, the corrupt replica is deleted.

It is possible to disable verification of checksums by passing `false` to the `setVerifyChecksum()` method on `FileSystem` before using the `open()` method to read a file. The same effect is possible from the shell by using the `-ignoreCrc` option with the `-get` or the equivalent `-copyToLocal` command. This feature is useful if you have a corrupt file that you want to inspect so you can decide what to do with it. For example, you might want to see whether it can be salvaged before you delete it.

You can find a file's checksum with `hadoop fs -checksum`. This is useful to check whether two files in HDFS have the same contents—something that *distcp* does, for example (see "Parallel Copying with distcp" on page 76).

LocalFileSystem

The Hadoop `LocalFileSystem` performs client-side checksumming. This means that when you write a file called *filename*, the filesystem client transparently creates a hidden file, *.filename.crc*, in the same directory containing the checksums for each chunk of the file. The chunk size is controlled by the `file.bytes-per-checksum` property, which defaults to 512 bytes. The chunk size is stored as metadata in the *.crc* file, so the file can be read back correctly even if the setting for the chunk size has changed. Checksums are verified when the file is read, and if an error is detected, `LocalFileSystem` throws a `ChecksumException`.

Checksums are fairly cheap to compute (in Java, they are implemented in native code), typically adding a few percent overhead to the time to read or write a file. For most applications, this is an acceptable price to pay for data integrity. It is, however, possible to disable checksums, which is typically done when the underlying filesystem supports checksums natively. This is accomplished by using `RawLocalFileSystem` in place of `LocalFileSystem`. To do this globally in an application, it suffices to remap the implementation for `file` URIs by setting the property `fs.file.impl` to the value `org.apache.hadoop.fs.RawLocalFileSystem`. Alternatively, you can directly create a `RawLocalFileSystem` instance, which may be useful if you want to disable checksum verification for only some reads, for example:

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs.initialize(null, conf);
```

ChecksumFileSystem

`LocalFileSystem` uses `ChecksumFileSystem` to do its work, and this class makes it easy to add checksumming to other (noncheckstummed) filesystems, as `ChecksumFileSystem` is just a wrapper around `FileSystem`. The general idiom is as follows:

```
FileSystem rawFs = ...
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

The underlying filesystem is called the *raw* filesystem, and may be retrieved using the `getRawFileSystem()` method on `ChecksumFileSystem`. `ChecksumFileSystem` has a few more useful methods for working with checksums, such as `getChecksumFile()` for getting the path of a checksum file for any file. Check the documentation for the others.

If an error is detected by `ChecksumFileSystem` when reading a file, it will call its `reportChecksumFailure()` method. The default implementation does nothing, but `LocalFileSystem` moves the offending file and its checksum to a side directory on the same device called *bad_files*. Administrators should periodically check for these bad files and take action on them.

Compression

File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.

There are many different compression formats, tools, and algorithms, each with different characteristics. Table 5-1 lists some of the more common ones that can be used with Hadoop.

Table 5-1. A summary of compression formats

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

[a] DEFLATE is a compression algorithm whose standard implementation is zlib. There is no commonly available command-line tool for producing files in DEFLATE format, as gzip is normally used. (Note that the gzip file format is DEFLATE with extra headers and a footer.) The *.deflate* filename extension is a Hadoop convention.

[b] However, LZO files are splittable if they have been indexed in a preprocessing step. See "Compression and Input Splits" on page 105.

All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. The tools listed in Table 5-1 typically give some control over this trade-off at compression time by offering nine different options: `-1` means optimize for speed, and `-9` means optimize for space. For example, the following command creates a compressed file *file.gz* using the fastest compression method:

```
% gzip -1 file
```

The different tools have very different compression characteristics. gzip is a general-purpose compressor and sits in the middle of the space/time trade-off. bzip2 compresses more effectively than gzip, but is slower. bzip2's decompression speed is faster than its compression speed, but it is still slower than the other formats. LZO, LZ4, and Snappy, on the other hand, all optimize for speed and are around an order of magnitude faster

than gzip, but compress less effectively. Snappy and LZ4 are also significantly faster than LZO for decompression.[1]

The "Splittable" column in Table 5-1 indicates whether the compression format supports splitting (that is, whether you can seek to any point in the stream and start reading from some point further on). Splittable compression formats are especially suitable for MapReduce; see "Compression and Input Splits" on page 105 for further discussion.

Codecs

A *codec* is the implementation of a compression-decompression algorithm. In Hadoop, a codec is represented by an implementation of the `CompressionCodec` interface. So, for example, `GzipCodec` encapsulates the compression and decompression algorithm for gzip. Table 5-2 lists the codecs that are available for Hadoop.

Table 5-2. Hadoop compression codecs

Compression format	Hadoop CompressionCodec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
LZ4	`org.apache.hadoop.io.compress.Lz4Codec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

The LZO libraries are GPL licensed and may not be included in Apache distributions, so for this reason the Hadoop codecs must be downloaded separately from Google (or GitHub, which includes bug fixes and more tools). The `LzopCodec`, which is compatible with the *lzop* tool, is essentially the LZO format with extra headers, and is the one you normally want. There is also an `LzoCodec` for the pure LZO format, which uses the *.lzo_deflate* filename extension (by analogy with DEFLATE, which is gzip without the headers).

Compressing and decompressing streams with CompressionCodec

`CompressionCodec` has two methods that allow you to easily compress or decompress data. To compress data being written to an output stream, use the `createOutputStream(OutputStream out)` method to create a `CompressionOutputStream` to which you write your uncompressed data to have it written in compressed form to the underlying stream. Conversely, to decompress data being read from an input stream,

1. For a comprehensive set of compression benchmarks, *jvm-compressor-benchmark* is a good reference for JVM-compatible libraries (including some native libraries).

call `createInputStream(InputStream in)` to obtain a `CompressionInputStream`, which allows you to read uncompressed data from the underlying stream.

`CompressionOutputStream` and `CompressionInputStream` are similar to `java.util.zip.DeflaterOutputStream` and `java.util.zip.DeflaterInputStream`, except that both of the former provide the ability to reset their underlying compressor or decompressor. This is important for applications that compress sections of the data stream as separate blocks, such as in a `SequenceFile`, described in "SequenceFile" on page 127.

Example 5-1 illustrates how to use the API to compress data read from standard input and write it to standard output.

Example 5-1. A program to compress data read from standard input and write it to standard output

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

The application expects the fully qualified name of the `CompressionCodec` implementation as the first command-line argument. We use `ReflectionUtils` to construct a new instance of the codec, then obtain a compression wrapper around `System.out`. Then we call the utility method `copyBytes()` on `IOUtils` to copy the input to the output, which is compressed by the `CompressionOutputStream`. Finally, we call `finish()` on `CompressionOutputStream`, which tells the compressor to finish writing to the compressed stream, but doesn't close the stream. We can try it out with the following command line, which compresses the string "Text" using the `StreamCompressor` program with the `GzipCodec`, then decompresses it from standard input using *gunzip*:

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec \
  | gunzip -
Text
```

Inferring CompressionCodecs using CompressionCodecFactory

If you are reading a compressed file, normally you can infer which codec to use by looking at its filename extension. A file ending in *.gz* can be read with `GzipCodec`, and so on. The extensions for each compression format are listed in Table 5-1.

`CompressionCodecFactory` provides a way of mapping a filename extension to a `CompressionCodec` using its `getCodec()` method, which takes a `Path` object for the file in question. Example 5-2 shows an application that uses this feature to decompress files.

Example 5-2. A program to decompress a compressed file using a codec inferred from the file's extension

```
public class FileDecompressor {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path inputPath = new Path(uri);
    CompressionCodecFactory factory = new CompressionCodecFactory(conf);
    CompressionCodec codec = factory.getCodec(inputPath);
    if (codec == null) {
      System.err.println("No codec found for " + uri);
      System.exit(1);
    }

    String outputUri =
        CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());

    InputStream in = null;
    OutputStream out = null;
    try {
      in = codec.createInputStream(fs.open(inputPath));
      out = fs.create(new Path(outputUri));
      IOUtils.copyBytes(in, out, conf);
    } finally {
      IOUtils.closeStream(in);
      IOUtils.closeStream(out);
    }
  }
}
```

Once the codec has been found, it is used to strip off the file suffix to form the output filename (via the `removeSuffix()` static method of `CompressionCodecFactory`). In this way, a file named *file.gz* is decompressed to *file* by invoking the program as follows:

```
% hadoop FileDecompressor file.gz
```

`CompressionCodecFactory` loads all the codecs in Table 5-2, except LZO, as well as any listed in the `io.compression.codecs` configuration property (Table 5-3). By default, the property is empty; you would need to alter it only if you have a custom codec that you wish to register (such as the externally hosted LZO codecs). Each codec knows its default filename extension, thus permitting `CompressionCodecFactory` to search through the registered codecs to find a match for the given extension (if any).

Table 5-3. Compression codec properties

Property name	Type	Default value	Description
`io.compression.codecs`	Comma-separated `Class` names		A list of additional `CompressionCodec` classes for compression/decompression

Native libraries

For performance, it is preferable to use a native library for compression and decompression. For example, in one test, using the native gzip libraries reduced decompression times by up to 50% and compression times by around 10% (compared to the built-in Java implementation). Table 5-4 shows the availability of Java and native implementations for each compression format. All formats have native implementations, but not all have a Java implementation (LZO, for example).

Table 5-4. Compression library implementations

Compression format	Java implementation?	Native implementation?
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	Yes
LZO	No	Yes
LZ4	No	Yes
Snappy	No	Yes

The Apache Hadoop binary tarball comes with prebuilt native compression binaries for 64-bit Linux, called *libhadoop.so*. For other platforms, you will need to compile the libraries yourself, following the *BUILDING.txt* instructions at the top level of the source tree.

The native libraries are picked up using the Java system property `java.library.path`. The *hadoop* script in the *etc/hadoop* directory sets this property for you, but if you don't use this script, you will need to set the property in your application.

By default, Hadoop looks for native libraries for the platform it is running on, and loads them automatically if they are found. This means you don't have to change any configuration settings to use the native libraries. In some circumstances, however, you may wish to disable use of native libraries, such as when you are debugging a compression-related problem. You can do this by setting the property `io.native.lib.available` to `false`, which ensures that the built-in Java equivalents will be used (if they are available).

CodecPool. If you are using a native library and you are doing a lot of compression or decompression in your application, consider using `CodecPool`, which allows you to

reuse compressors and decompressors, thereby amortizing the cost of creating these objects.

The code in Example 5-3 shows the API, although in this program, which creates only a single `Compressor`, there is really no need to use a pool.

Example 5-3. A program to compress data read from standard input and write it to standard output using a pooled compressor

```
public class PooledStreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
        ReflectionUtils.newInstance(codecClass, conf);
    Compressor compressor = null;
    try {
      compressor = CodecPool.getCompressor(codec);
      CompressionOutputStream out =
          codec.createOutputStream(System.out, compressor);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    } finally {
      CodecPool.returnCompressor(compressor);
    }
  }
}
```

We retrieve a `Compressor` instance from the pool for a given `CompressionCodec`, which we use in the codec's overloaded `createOutputStream()` method. By using a `finally` block, we ensure that the compressor is returned to the pool even if there is an `IOException` while copying the bytes between the streams.

Compression and Input Splits

When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting. Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 128 MB, the file will be stored as eight blocks, and a MapReduce job using this file as input will create eight input splits, each processed independently as input to a separate map task.

Imagine now that the file is a gzip-compressed file whose compressed size is 1 GB. As before, HDFS will store the file as eight blocks. However, creating a split for each block won't work, because it is impossible to start reading at an arbitrary point in the gzip stream and therefore impossible for a map task to read its split independently of the

others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The problem is that the start of each block is not distinguished in any way that would allow a reader positioned at an arbitrary point in the stream to advance to the beginning of the next block, thereby synchronizing itself with the stream. For this reason, gzip does not support splitting.

In this case, MapReduce will do the right thing and not try to split the gzipped file, since it knows that the input is gzip-compressed (by looking at the filename extension) and that gzip does not support splitting. This will work, but at the expense of locality: a single map will process the eight HDFS blocks, most of which will not be local to the map. Also, with fewer maps, the job is less granular and so may take longer to run.

If the file in our hypothetical example were an LZO file, we would have the same problem because the underlying compression format does not provide a way for a reader to synchronize itself with the stream. However, it is possible to preprocess LZO files using an indexer tool that comes with the Hadoop LZO libraries, which you can obtain from the Google and GitHub sites listed in "Codecs" on page 101. The tool builds an index of split points, effectively making them splittable when the appropriate MapReduce input format is used.

A bzip2 file, on the other hand, does provide a synchronization marker between blocks (a 48-bit approximation of pi), so it does support splitting. (Table 5-1 lists whether each compression format supports splitting.)

Which Compression Format Should I Use?

Hadoop applications process large datasets, so you should strive to take advantage of compression. Which compression format you use depends on such considerations as file size, format, and the tools you are using for processing. Here are some suggestions, arranged roughly in order of most to least effective:

- Use a container file format such as sequence files (see the section on page 127), Avro datafiles (see the section on page 352), ORCFiles (see the section on page 136), or Parquet files (see the section on page 370), all of which support both compression and splitting. A fast compressor such as LZO, LZ4, or Snappy is generally a good choice.
- Use a compression format that supports splitting, such as bzip2 (although bzip2 is fairly slow), or one that can be indexed to support splitting, such as LZO.
- Split the file into chunks in the application, and compress each chunk separately using any supported compression format (it doesn't matter whether it is splittable). In this case, you should choose the chunk size so that the compressed chunks are approximately the size of an HDFS block.

- Store the files uncompressed.

For large files, you should *not* use a compression format that does not support splitting on the whole file, because you lose locality and make MapReduce applications very inefficient.

Using Compression in MapReduce

As described in "Inferring CompressionCodecs using CompressionCodecFactory" on page 102, if your input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use.

In order to compress the output of a MapReduce job, in the job configuration, set the `mapreduce.output.fileoutputformat.compress` property to `true` and set the `mapreduce.output.fileoutputformat.compress.codec` property to the classname of the compression codec you want to use. Alternatively, you can use the static convenience methods on `FileOutputFormat` to set these properties, as shown in Example 5-4.

Example 5-4. Application to run the maximum temperature job producing compressed output

```
public class MaxTemperatureWithCompression {

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

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);

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

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

    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

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

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

```
  }
}
```

We run the program over compressed input (which doesn't have to use the same compression format as the output, although it does in this example) as follows:

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output
```

Each part of the final output is compressed; in this case, there is a single part:

```
% gunzip -c output/part-r-00000.gz
1949    111
1950    22
```

If you are emitting sequence files for your output, you can set the `mapreduce.output.fileoutputformat.compress.type` property to control the type of compression to use. The default is `RECORD`, which compresses individual records. Changing this to `BLOCK`, which compresses groups of records, is recommended because it compresses better (see "The SequenceFile format" on page 133).

There is also a static convenience method on `SequenceFileOutputFormat` called `setOutputCompressionType()` to set this property.

The configuration properties to set compression for MapReduce job outputs are summarized in Table 5-5. If your MapReduce driver uses the `Tool` interface (described in "GenericOptionsParser, Tool, and ToolRunner" on page 148), you can pass any of these properties to the program on the command line, which may be more convenient than modifying your program to hardcode the compression properties.

Table 5-5. MapReduce compression properties

Property name	Type	Default value	Description
`mapreduce.output.fileoutputformat.compress`	`boolean`	`false`	Whether to compress outputs
`mapreduce.output.fileoutputformat.compress.codec`	`Class name`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for outputs
`mapreduce.output.fileoutputformat.compress.type`	`String`	`RECORD`	The type of compression to use for sequence file outputs: `NONE`, `RECORD`, or `BLOCK`

Compressing map output

Even if your MapReduce application reads and writes uncompressed data, it may benefit from compressing the intermediate output of the map phase. The map output is written to disk and transferred across the network to the reducer nodes, so by using a fast

compressor such as LZO, LZ4, or Snappy, you can get performance gains simply because the volume of data to transfer is reduced. The configuration properties to enable compression for map outputs and to set the compression format are shown in Table 5-6.

Table 5-6. Map output compression properties

Property name	Type	Default value	Description
`mapreduce.map.output.compress`	`boolean`	`false`	Whether to compress map outputs
`mapreduce.map.output.compress.codec`	`Class`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for map outputs

Here are the lines to add to enable gzip map output compression in your job (using the new API):

```
Configuration conf = new Configuration();
conf.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, GzipCodec.class,
    CompressionCodec.class);
Job job = new Job(conf);
```

In the old API (see Appendix D), there are convenience methods on the `JobConf` object for doing the same thing:

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

Serialization

Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. *Deserialization* is the reverse process of turning a byte stream back into a series of structured objects.

Serialization is used in two quite distinct areas of distributed data processing: for interprocess communication and for persistent storage.

In Hadoop, interprocess communication between nodes in the system is implemented using *remote procedure calls* (RPCs). The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message. In general, it is desirable that an RPC serialization format is:

Compact
: A compact format makes the best use of network bandwidth, which is the most scarce resource in a data center.

Fast
: Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.

Extensible
: Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers. For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.

Interoperable
: For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

On the face of it, the data format chosen for persistent storage would have different requirements from a serialization framework. After all, the lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written. But it turns out, the four desirable properties of an RPC's serialization format are also crucial for a persistent storage format. We want the storage format to be compact (to make efficient use of storage space), fast (so the overhead in reading or writing terabytes of data is minimal), extensible (so we can transparently read data written in an older format), and interoperable (so we can read or write persistent data using different languages).

Hadoop uses its own serialization format, Writables, which is certainly compact and fast, but not so easy to extend or use from languages other than Java. Because Writables are central to Hadoop (most MapReduce programs use them for their key and value types), we look at them in some depth in the next three sections, before looking at some of the other serialization frameworks supported in Hadoop. Avro (a serialization system that was designed to overcome some of the limitations of Writables) is covered in Chapter 12.

The Writable Interface

The `Writable` interface defines two methods—one for writing its state to a `DataOutput` binary stream and one for reading its state from a `DataInput` binary stream:

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
```

CHAPTER 10
Setting Up a Hadoop Cluster

This chapter explains how to set up Hadoop to run on a cluster of machines. Running HDFS, MapReduce, and YARN on a single machine is great for learning about these systems, but to do useful work, they need to run on multiple nodes.

There are a few options when it comes to getting a Hadoop cluster, from building your own, to running on rented hardware or using an offering that provides Hadoop as a hosted service in the cloud. The number of hosted options is too large to list here, but even if you choose to build a Hadoop cluster yourself, there are still a number of installation options:

Apache tarballs
: The Apache Hadoop project and related projects provide binary (and source) tarballs for each release. Installation from binary tarballs gives you the most flexibility but entails the most amount of work, since you need to decide on where the installation files, configuration files, and logfiles are located on the filesystem, set their file permissions correctly, and so on.

Packages
: RPM and Debian packages are available from the Apache Bigtop project, as well as from all the Hadoop vendors. Packages bring a number of advantages over tarballs: they provide a consistent filesystem layout, they are tested together as a stack (so you know that the versions of Hadoop and Hive, say, will work together), and they work well with configuration management tools like Puppet.

Hadoop cluster management tools
: Cloudera Manager and Apache Ambari are examples of dedicated tools for installing and managing a Hadoop cluster over its whole lifecycle. They provide a simple web UI, and are the recommended way to set up a Hadoop cluster for most users and operators. These tools encode a lot of operator knowledge about running Hadoop. For example, they use heuristics based on the hardware profile (among

other factors) to choose good defaults for Hadoop configuration settings. For more complex setups, like HA, or secure Hadoop, the management tools provide well-tested wizards for getting a working cluster in a short amount of time. Finally, they add extra features that the other installation options don't offer, such as unified monitoring and log search, and rolling upgrades (so you can upgrade the cluster without experiencing downtime).

This chapter and the next give you enough information to set up and operate your own basic cluster, but even if you are using Hadoop cluster management tools or a service in which a lot of the routine setup and maintenance are done for you, these chapters still offer valuable information about how Hadoop works from an operations point of view. For more in-depth information, I highly recommend *Hadoop Operations* by Eric Sammer (O'Reilly, 2012).

Cluster Specification

Hadoop is designed to run on commodity hardware. That means that you are not tied to expensive, proprietary offerings from a single vendor; rather, you can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster.

"Commodity" does not mean "low-end." Low-end machines often have cheap components, which have higher failure rates than more expensive (but still commodity-class) machines. When you are operating tens, hundreds, or thousands of machines, cheap components turn out to be a false economy, as the higher failure rate incurs a greater maintenance cost. On the other hand, large database-class machines are not recommended either, since they don't score well on the price/performance curve. And even though you would need fewer of them to build a cluster of comparable performance to one built of mid-range commodity hardware, when one did fail, it would have a bigger impact on the cluster because a larger proportion of the cluster hardware would be unavailable.

Hardware specifications rapidly become obsolete, but for the sake of illustration, a typical choice of machine for running an HDFS datanode and a YARN node manager in 2014 would have had the following specifications:

Processor
: Two hex/octo-core 3 GHz CPUs

Memory
: 64–512 GB ECC RAM[1]

1. ECC memory is strongly recommended, as several Hadoop users have reported seeing many checksum errors when using non-ECC memory on Hadoop clusters.

Storage
: 12–24 × 1–4 TB SATA disks

Network
: Gigabit Ethernet with link aggregation

Although the hardware specification for your cluster will assuredly be different, Hadoop is designed to use multiple cores and disks, so it will be able to take full advantage of more powerful hardware.

Why Not Use RAID?

HDFS clusters do not benefit from using RAID (redundant array of independent disks) for datanode storage (although RAID is recommended for the namenode's disks, to protect against corruption of its metadata). The redundancy that RAID provides is not needed, since HDFS handles it by replication between nodes.

Furthermore, RAID striping (RAID 0), which is commonly used to increase performance, turns out to be *slower* than the JBOD (just a bunch of disks) configuration used by HDFS, which round-robins HDFS blocks between all disks. This is because RAID 0 read and write operations are limited by the speed of the slowest-responding disk in the RAID array. In JBOD, disk operations are independent, so the average speed of operations is greater than that of the slowest disk. Disk performance often shows considerable variation in practice, even for disks of the same model. In some benchmarking carried out on a Yahoo! cluster, JBOD performed 10% faster than RAID 0 in one test (Gridmix) and 30% better in another (HDFS write throughput).

Finally, if a disk fails in a JBOD configuration, HDFS can continue to operate without the failed disk, whereas with RAID, failure of a single disk causes the whole array (and hence the node) to become unavailable.

Cluster Sizing

How large should your cluster be? There isn't an exact answer to this question, but the beauty of Hadoop is that you can start with a small cluster (say, 10 nodes) and grow it as your storage and computational needs grow. In many ways, a better question is this: how fast does your cluster need to grow? You can get a good feel for this by considering storage capacity.

For example, if your data grows by 1 TB a day and you have three-way HDFS replication, you need an additional 3 TB of raw storage per day. Allow some room for intermediate files and logfiles (around 30%, say), and this is in the range of one (2014-vintage) machine per week. In practice, you wouldn't buy a new machine each week and add it to the cluster. The value of doing a back-of-the-envelope calculation like this is that it gives

you a feel for how big your cluster should be. In this example, a cluster that holds two years' worth of data needs 100 machines.

Master node scenarios

Depending on the size of the cluster, there are various configurations for running the master daemons: the namenode, secondary namenode, resource manager, and history server. For a small cluster (on the order of 10 nodes), it is usually acceptable to run the namenode and the resource manager on a single master machine (as long as at least one copy of the namenode's metadata is stored on a remote filesystem). However, as the cluster gets larger, there are good reasons to separate them.

The namenode has high memory requirements, as it holds file and block metadata for the entire namespace in memory. The secondary namenode, although idle most of the time, has a comparable memory footprint to the primary when it creates a checkpoint. (This is explained in detail in "The filesystem image and edit log" on page 318.) For filesystems with a large number of files, there may not be enough physical memory on one machine to run both the primary and secondary namenode.

Aside from simple resource requirements, the main reason to run masters on separate machines is for high availability. Both HDFS and YARN support configurations where they can run masters in active-standby pairs. If the active master fails, then the standby, running on separate hardware, takes over with little or no interruption to the service. In the case of HDFS, the standby performs the checkpointing function of the secondary namenode (so you don't need to run a standby and a secondary namenode).

Configuring and running Hadoop HA is not covered in this book. Refer to the Hadoop website or vendor documentation for details.

Network Topology

A common Hadoop cluster architecture consists of a two-level network topology, as illustrated in Figure 10-1. Typically there are 30 to 40 servers per rack (only 3 are shown in the diagram), with a 10 Gb switch for the rack and an uplink to a core switch or router (at least 10 Gb or better). The salient point is that the aggregate bandwidth between nodes on the same rack is much greater than that between nodes on different racks.

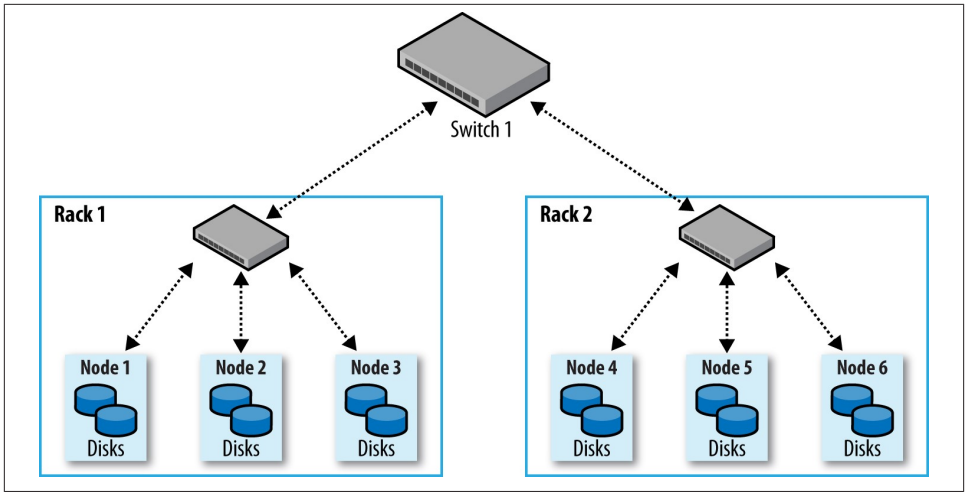

Figure 10-1. Typical two-level network architecture for a Hadoop cluster

Rack awareness

To get maximum performance out of Hadoop, it is important to configure Hadoop so that it knows the topology of your network. If your cluster runs on a single rack, then there is nothing more to do, since this is the default. However, for multirack clusters, you need to map nodes to racks. This allows Hadoop to prefer within-rack transfers (where there is more bandwidth available) to off-rack transfers when placing MapReduce tasks on nodes. HDFS will also be able to place replicas more intelligently to trade off performance and resilience.

Network locations such as nodes and racks are represented in a tree, which reflects the network "distance" between locations. The namenode uses the network location when determining where to place block replicas (see "Network Topology and Hadoop" on page 70); the MapReduce scheduler uses network location to determine where the closest replica is for input to a map task.

For the network in Figure 10-1, the rack topology is described by two network locations —say, */switch1/rack1* and */switch1/rack2*. Because there is only one top-level switch in this cluster, the locations can be simplified to */rack1* and */rack2*.

The Hadoop configuration must specify a map between node addresses and network locations. The map is described by a Java interface, `DNSToSwitchMapping`, whose signature is:

```
public interface DNSToSwitchMapping {
  public List<String> resolve(List<String> names);
}
```

The `names` parameter is a list of IP addresses, and the return value is a list of corresponding network location strings. The `net.topology.node.switch.mapping.impl` configuration property defines an implementation of the `DNSToSwitchMapping` interface that the namenode and the resource manager use to resolve worker node network locations.

For the network in our example, we would map *node1*, *node2*, and *node3* to */rack1*, and *node4*, *node5*, and *node6* to */rack2*.

Most installations don't need to implement the interface themselves, however, since the default implementation is `ScriptBasedMapping`, which runs a user-defined script to determine the mapping. The script's location is controlled by the property `net.topology.script.file.name`. The script must accept a variable number of arguments that are the hostnames or IP addresses to be mapped, and it must emit the corresponding network locations to standard output, separated by whitespace. The Hadoop wiki has an example.

If no script location is specified, the default behavior is to map all nodes to a single network location, called */default-rack*.

Cluster Setup and Installation

This section describes how to install and configure a basic Hadoop cluster from scratch using the Apache Hadoop distribution on a Unix operating system. It provides background information on the things you need to think about when setting up Hadoop. For a production installation, most users and operators should consider one of the Hadoop cluster management tools listed at the beginning of this chapter.

Installing Java

Hadoop runs on both Unix and Windows operating systems, and requires Java to be installed. For a production installation, you should select a combination of operating system, Java, and Hadoop that has been certified by the vendor of the Hadoop distribution you are using. There is also a page on the Hadoop wiki that lists combinations that community members have run with success.

Creating Unix User Accounts

It's good practice to create dedicated Unix user accounts to separate the Hadoop processes from each other, and from other services running on the same machine. The HDFS, MapReduce, and YARN services are usually run as separate users, named `hdfs`, `mapred`, and `yarn`, respectively. They all belong to the same `hadoop` group.

Installing Hadoop

Download Hadoop from the Apache Hadoop releases page, and unpack the contents of the distribution in a sensible location, such as */usr/local* (*/opt* is another standard choice; note that Hadoop should not be installed in a user's home directory, as that may be an NFS-mounted directory):

```
% cd /usr/local
% sudo tar xzf hadoop-x.y.z.tar.gz
```

You also need to change the owner of the Hadoop files to be the `hadoop` user and group:

```
% sudo chown -R hadoop:hadoop hadoop-x.y.z
```

It's convenient to put the Hadoop binaries on the shell path too:

```
% export HADOOP_HOME=/usr/local/hadoop-x.y.z
% export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

Configuring SSH

The Hadoop control scripts (but not the daemons) rely on SSH to perform cluster-wide operations. For example, there is a script for stopping and starting all the daemons in the cluster. Note that the control scripts are optional—cluster-wide operations can be performed by other mechanisms, too, such as a distributed shell or dedicated Hadoop management applications.

To work seamlessly, SSH needs to be set up to allow passwordless login for the `hdfs` and `yarn` users from machines in the cluster.[2] The simplest way to achieve this is to generate a public/private key pair and place it in an NFS location that is shared across the cluster.

First, generate an RSA key pair by typing the following. You need to do this twice, once as the `hdfs` user and once as the `yarn` user:

```
% ssh-keygen -t rsa -f ~/.ssh/id_rsa
```

Even though we want passwordless logins, keys without passphrases are not considered good practice (it's OK to have an empty passphrase when running a local pseudo-distributed cluster, as described in Appendix A), so we specify a passphrase when prompted for one. We use *ssh-agent* to avoid the need to enter a password for each connection.

The private key is in the file specified by the `-f` option, *~/.ssh/id_rsa*, and the public key is stored in a file with the same name but with *.pub* appended, *~/.ssh/id_rsa.pub*.

2. The `mapred` user doesn't use SSH, as in Hadoop 2 and later, the only MapReduce daemon is the job history server.

Next, we need to make sure that the public key is in the *~/.ssh/authorized_keys* file on all the machines in the cluster that we want to connect to. If the users' home directories are stored on an NFS filesystem, the keys can be shared across the cluster by typing the following (first as `hdfs` and then as `yarn`):

```
% cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

If the home directory is not shared using NFS, the public keys will need to be shared by some other means (such as *ssh-copy-id*).

Test that you can SSH from the master to a worker machine by making sure *ssh-agent* is running,[3] and then run *ssh-add* to store your passphrase. You should be able to SSH to a worker without entering the passphrase again.

Configuring Hadoop

Hadoop must have its configuration set appropriately to run in distributed mode on a cluster. The important configuration settings to achieve this are discussed in "Hadoop Configuration" on page 292.

Formatting the HDFS Filesystem

Before it can be used, a brand-new HDFS installation needs to be formatted. The formatting process creates an empty filesystem by creating the storage directories and the initial versions of the namenode's persistent data structures. Datanodes are not involved in the initial formatting process, since the namenode manages all of the filesystem's metadata, and datanodes can join or leave the cluster dynamically. For the same reason, you don't need to say how large a filesystem to create, since this is determined by the number of datanodes in the cluster, which can be increased as needed, long after the filesystem is formatted.

Formatting HDFS is a fast operation. Run the following command as the `hdfs` user:

```
% hdfs namenode -format
```

Starting and Stopping the Daemons

Hadoop comes with scripts for running commands and starting and stopping daemons across the whole cluster. To use these scripts (which can be found in the *sbin* directory), you need to tell Hadoop which machines are in the cluster. There is a file for this purpose, called *slaves*, which contains a list of the machine hostnames or IP addresses, one per line. The *slaves* file lists the machines that the datanodes and node managers should run on. It resides in Hadoop's configuration directory, although it may be placed elsewhere

3. See its man page for instructions on how to start *ssh-agent*.

(and given another name) by changing the `HADOOP_SLAVES` setting in *hadoop-env.sh*. Also, this file does not need to be distributed to worker nodes, since they are used only by the control scripts running on the namenode or resource manager.

The HDFS daemons are started by running the following command as the `hdfs` user:

```
% start-dfs.sh
```

The machine (or machines) that the namenode and secondary namenode run on is determined by interrogating the Hadoop configuration for their hostnames. For example, the script finds the namenode's hostname by executing the following:

```
% hdfs getconf -namenodes
```

By default, this finds the namenode's hostname from `fs.defaultFS`. In slightly more detail, the *start-dfs.sh* script does the following:

- Starts a namenode on each machine returned by executing `hdfs getconf -namenodes`[4]
- Starts a datanode on each machine listed in the *slaves* file
- Starts a secondary namenode on each machine returned by executing `hdfs getconf -secondarynamenodes`

The YARN daemons are started in a similar way, by running the following command as the `yarn` user on the machine hosting the resource manager:

```
% start-yarn.sh
```

In this case, the resource manager is always run on the machine from which the *start-yarn.sh* script was run. More specifically, the script:

- Starts a resource manager on the local machine
- Starts a node manager on each machine listed in the *slaves* file

Also provided are *stop-dfs.sh* and *stop-yarn.sh* scripts to stop the daemons started by the corresponding start scripts.

These scripts start and stop Hadoop daemons using the *hadoop-daemon.sh* script (or the *yarn-daemon.sh* script, in the case of YARN). If you use the aforementioned scripts, you shouldn't call *hadoop-daemon.sh* directly. But if you need to control Hadoop daemons from another system or from your own scripts, the *hadoop-daemon.sh* script is a good integration point. Likewise, *hadoop-daemons.sh* (with an "s") is handy for starting the same daemon on a set of hosts.

4. There can be more than one namenode when running HDFS HA.

Finally, there is only one MapReduce daemon—the job history server, which is started as follows, as the `mapred` user:

```
% mr-jobhistory-daemon.sh start historyserver
```

Creating User Directories

Once you have a Hadoop cluster up and running, you need to give users access to it. This involves creating a home directory for each user and setting ownership permissions on it:

```
% hadoop fs -mkdir /user/username
% hadoop fs -chown username:username /user/username
```

This is a good time to set space limits on the directory. The following sets a 1 TB limit on the given user directory:

```
% hdfs dfsadmin -setSpaceQuota 1t /user/username
```

Hadoop Configuration

There are a handful of files for controlling the configuration of a Hadoop installation; the most important ones are listed in Table 10-1.

Table 10-1. Hadoop configuration files

Filename	Format	Description
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop
mapred-env.sh	Bash script	Environment variables that are used in the scripts to run MapReduce (overrides variables set in *hadoop-env.sh*)
yarn-env.sh	Bash script	Environment variables that are used in the scripts to run YARN (overrides variables set in *hadoop-env.sh*)
core-site.xml	Hadoop configuration XML	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS, MapReduce, and YARN
hdfs-site.xml	Hadoop configuration XML	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes
mapred-site.xml	Hadoop configuration XML	Configuration settings for MapReduce daemons: the job history server
yarn-site.xml	Hadoop configuration XML	Configuration settings for YARN daemons: the resource manager, the web app proxy server, and the node managers
slaves	Plain text	A list of machines (one per line) that each run a datanode and a node manager
hadoop-metrics2 .properties	Java properties	Properties for controlling how metrics are published in Hadoop (see "Metrics and JMX" on page 331)
log4j.properties	Java properties	Properties for system logfiles, the namenode audit log, and the task log for the task JVM process ("Hadoop Logs" on page 172)

Filename	Format	Description
hadoop-policy.xml	Hadoop configuration XML	Configuration settings for access control lists when running Hadoop in secure mode

These files are all found in the *etc/hadoop* directory of the Hadoop distribution. The configuration directory can be relocated to another part of the filesystem (outside the Hadoop installation, which makes upgrades marginally easier) as long as daemons are started with the `--config` option (or, equivalently, with the `HADOOP_CONF_DIR` environment variable set) specifying the location of this directory on the local filesystem.

Configuration Management

Hadoop does not have a single, global location for configuration information. Instead, each Hadoop node in the cluster has its own set of configuration files, and it is up to administrators to ensure that they are kept in sync across the system. There are parallel shell tools that can help do this, such as *dsh* or *pdsh*. This is an area where Hadoop cluster management tools like Cloudera Manager and Apache Ambari really shine, since they take care of propagating changes across the cluster.

Hadoop is designed so that it is possible to have a single set of configuration files that are used for all master and worker machines. The great advantage of this is simplicity, both conceptually (since there is only one configuration to deal with) and operationally (as the Hadoop scripts are sufficient to manage a single configuration setup).

For some clusters, the one-size-fits-all configuration model breaks down. For example, if you expand the cluster with new machines that have a different hardware specification from the existing ones, you need a different configuration for the new machines to take advantage of their extra resources.

In these cases, you need to have the concept of a *class* of machine and maintain a separate configuration for each class. Hadoop doesn't provide tools to do this, but there are several excellent tools for doing precisely this type of configuration management, such as Chef, Puppet, CFEngine, and Bcfg2.

For a cluster of any size, it can be a challenge to keep all of the machines in sync. Consider what happens if the machine is unavailable when you push out an update. Who ensures it gets the update when it becomes available? This is a big problem and can lead to divergent installations, so even if you use the Hadoop control scripts for managing Hadoop, it may be a good idea to use configuration management tools for maintaining the cluster. These tools are also excellent for doing regular maintenance, such as patching security holes and updating system packages.

CHAPTER 16
Pig

Apache Pig raises the level of abstraction for processing large datasets. MapReduce allows you, as the programmer, to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MapReduce stages, can be a challenge. With Pig, the data structures are much richer, typically being multivalued and nested, and the transformations you can apply to the data are much more powerful. They include joins, for example, which are not for the faint of heart in MapReduce.

Pig is made up of two pieces:

- The language used to express data flows, called *Pig Latin*.
- The execution environment to run Pig Latin programs. There are currently two environments: local execution in a single JVM and distributed execution on a Hadoop cluster.

A Pig Latin program is made up of a series of operations, or transformations, that are applied to the input data to produce output. Taken as a whole, the operations describe a data flow, which the Pig execution environment translates into an executable representation and then runs. Under the covers, Pig turns the transformations into a series of MapReduce jobs, but as a programmer you are mostly unaware of this, which allows you to focus on the data rather than the nature of the execution.

Pig is a scripting language for exploring large datasets. One criticism of MapReduce is that the development cycle is very long. Writing the mappers and reducers, compiling and packaging the code, submitting the job(s), and retrieving the results is a time-consuming business, and even with Streaming, which removes the compile and package step, the experience is still involved. Pig's sweet spot is its ability to process terabytes of data in response to a half-dozen lines of Pig Latin issued from the console. Indeed, it was created at Yahoo! to make it easier for researchers and engineers to mine the huge

datasets there. Pig is very supportive of a programmer writing a query, since it provides several commands for introspecting the data structures in your program as it is written. Even more useful, it can perform a sample run on a representative subset of your input data, so you can see whether there are errors in the processing before unleashing it on the full dataset.

Pig was designed to be extensible. Virtually all parts of the processing path are customizable: loading, storing, filtering, grouping, and joining can all be altered by user-defined functions (UDFs). These functions operate on Pig's nested data model, so they can integrate very deeply with Pig's operators. As another benefit, UDFs tend to be more reusable than the libraries developed for writing MapReduce programs.

In some cases, Pig doesn't perform as well as programs written in MapReduce. However, the gap is narrowing with each release, as the Pig team implements sophisticated algorithms for applying Pig's relational operators. It's fair to say that unless you are willing to invest a lot of effort optimizing Java MapReduce code, writing queries in Pig Latin will save you time.

Installing and Running Pig

Pig runs as a client-side application. Even if you want to run Pig on a Hadoop cluster, there is nothing extra to install on the cluster: Pig launches jobs and interacts with HDFS (or other Hadoop filesystems) from your workstation.

Installation is straightforward. Download a stable release from *http://pig.apache.org/releases.html*, and unpack the tarball in a suitable place on your workstation:

```
% tar xzf pig-x.y.z.tar.gz
```

It's convenient to add Pig's binary directory to your command-line path. For example:

```
% export PIG_HOME=~/sw/pig-x.y.z
% export PATH=$PATH:$PIG_HOME/bin
```

You also need to set the `JAVA_HOME` environment variable to point to a suitable Java installation.

Try typing `pig -help` to get usage instructions.

Execution Types

Pig has two execution types or modes: local mode and MapReduce mode. Execution modes for Apache Tez and Spark (see Chapter 19) were both under development at the time of writing. Both promise significant performance gains over MapReduce mode, so try them if they are available in the version of Pig you are using.

Local mode

In local mode, Pig runs in a single JVM and accesses the local filesystem. This mode is suitable only for small datasets and when trying out Pig.

The execution type is set using the `-x` or `-exectype` option. To run in local mode, set the option to `local`:

```
% pig -x local
grunt>
```

This starts Grunt, the Pig interactive shell, which is discussed in more detail shortly.

MapReduce mode

In MapReduce mode, Pig translates queries into MapReduce jobs and runs them on a Hadoop cluster. The cluster may be a pseudo- or fully distributed cluster. MapReduce mode (with a fully distributed cluster) is what you use when you want to run Pig on large datasets.

To use MapReduce mode, you first need to check that the version of Pig you downloaded is compatible with the version of Hadoop you are using. Pig releases will only work against particular versions of Hadoop; this is documented in the release notes.

Pig honors the `HADOOP_HOME` environment variable for finding which Hadoop client to run. However, if it is not set, Pig will use a bundled copy of the Hadoop libraries. Note that these may not match the version of Hadoop running on your cluster, so it is best to explicitly set `HADOOP_HOME`.

Next, you need to point Pig at the cluster's namenode and resource manager. If the installation of Hadoop at `HADOOP_HOME` is already configured for this, then there is nothing more to do. Otherwise, you can set `HADOOP_CONF_DIR` to a directory containing the Hadoop site file (or files) that define `fs.defaultFS`, `yarn.resourcemanager.address`, and `mapreduce.framework.name` (the latter should be set to `yarn`).

Alternatively, you can set these properties in the *pig.properties* file in Pig's *conf* directory (or the directory specified by `PIG_CONF_DIR`). Here's an example for a pseudo-distributed setup:

```
fs.defaultFS=hdfs://localhost/
mapreduce.framework.name=yarn
yarn.resourcemanager.address=localhost:8032
```

Once you have configured Pig to connect to a Hadoop cluster, you can launch Pig, setting the `-x` option to `mapreduce` or omitting it entirely, as MapReduce mode is the default. We've used the `-brief` option to stop timestamps from being logged:

```
% pig -brief
Logging error messages to: /Users/tom/pig_1414246949680.log
Default bootup file /Users/tom/.pigbootup not found
```

```
Connecting to hadoop file system at: hdfs://localhost/
grunt>
```

As you can see from the output, Pig reports the filesystem (but not the YARN resource manager) that it has connected to.

In MapReduce mode, you can optionally enable *auto-local mode* (by setting `pig.auto.local.enabled` to `true`), which is an optimization that runs small jobs locally if the input is less than 100 MB (set by `pig.auto.local.input.maxbytes`, default 100,000,000) and no more than one reducer is being used.

Running Pig Programs

There are three ways of executing Pig programs, all of which work in both local and MapReduce mode:

Script
: Pig can run a script file that contains Pig commands. For example, `pig script.pig` runs the commands in the local file *script.pig*. Alternatively, for very short scripts, you can use the `-e` option to run a script specified as a string on the command line.

Grunt
: Grunt is an interactive shell for running Pig commands. Grunt is started when no file is specified for Pig to run and the `-e` option is not used. It is also possible to run Pig scripts from within Grunt using `run` and `exec`.

Embedded
: You can run Pig programs from Java using the `PigServer` class, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use `PigRunner`.

Grunt

Grunt has line-editing facilities like those found in GNU Readline (used in the bash shell and many other command-line applications). For instance, the Ctrl-E key combination will move the cursor to the end of the line. Grunt remembers command history, too,[1] and you can recall lines in the history buffer using Ctrl-P or Ctrl-N (for previous and next), or equivalently, the up or down cursor keys.

Another handy feature is Grunt's completion mechanism, which will try to complete Pig Latin keywords and functions when you press the Tab key. For example, consider the following incomplete line:

```
grunt> a = foreach b ge
```

1. History is stored in a file called *.pig_history* in your home directory.

If you press the Tab key at this point, `ge` will expand to `generate`, a Pig Latin keyword:

```
grunt> a = foreach b generate
```

You can customize the completion tokens by creating a file named *autocomplete* and placing it on Pig's classpath (such as in the *conf* directory in Pig's *install* directory) or in the directory you invoked Grunt from. The file should have one token per line, and tokens must not contain any whitespace. Matching is case sensitive. It can be very handy to add commonly used file paths (especially because Pig does not perform filename completion) or the names of any user-defined functions you have created.

You can get a list of commands using the `help` command. When you've finished your Grunt session, you can exit with the `quit` command, or the equivalent shortcut `\q`.

Pig Latin Editors

There are Pig Latin syntax highlighters available for a variety of editors, including Eclipse, IntelliJ IDEA, Vim, Emacs, and TextMate. Details are available on the Pig wiki.

Many Hadoop distributions come with the Hue web interface, which has a Pig script editor and launcher.

An Example

Let's look at a simple example by writing the program to calculate the maximum recorded temperature by year for the weather dataset in Pig Latin (just like we did using MapReduce in Chapter 2). The complete program is only a few lines long:

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

To explore what's going on, we'll use Pig's Grunt interpreter, which allows us to enter lines and interact with the program to understand what it's doing. Start up Grunt in local mode, and then enter the first line of the Pig script:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year:chararray, temperature:int, quality:int);
```

For simplicity, the program assumes that the input is tab-delimited text, with each line having just year, temperature, and quality fields. (Pig actually has more flexibility than this with regard to the input formats it accepts, as we'll see later.) This line describes the input data we want to process. The `year:chararray` notation describes the field's name

and type; `chararray` is like a Java `String`, and an `int` is like a Java `int`. The `LOAD` operator takes a URI argument; here we are just using a local file, but we could refer to an HDFS URI. The `AS` clause (which is optional) gives the fields names to make it convenient to refer to them in subsequent statements.

The result of the `LOAD` operator, and indeed any operator in Pig Latin, is a *relation*, which is just a set of tuples. A *tuple* is just like a row of data in a database table, with multiple fields in a particular order. In this example, the `LOAD` function produces a set of (year, temperature, quality) tuples that are present in the input file. We write a relation with one tuple per line, where tuples are represented as comma-separated items in parentheses:

```
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
```

Relations are given names, or *aliases*, so they can be referred to. This relation is given the `records` alias. We can examine the contents of an alias using the `DUMP` operator:

```
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

We can also see the structure of a relation—the relation's *schema*—using the `DESCRIBE` operator on the relation's alias:

```
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
```

This tells us that `records` has three fields, with aliases `year`, `temperature`, and `quality`, which are the names we gave them in the `AS` clause. The fields have the types given to them in the `AS` clause, too. We examine types in Pig in more detail later.

The second statement removes records that have a missing temperature (indicated by a value of 9999) or an unsatisfactory quality reading. For this small dataset, no records are filtered out:

```
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>>   quality IN (0, 1, 4, 5, 9);
grunt> DUMP filtered_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

The third statement uses the GROUP function to group the records relation by the year field. Let's use DUMP to see what it produces:

```
grunt> grouped_records = GROUP filtered_records BY year;
grunt> DUMP grouped_records;
(1949,{(1949,78,1),(1949,111,1)})
(1950,{(1950,-11,1),(1950,22,1),(1950,0,1)})
```

We now have two rows, or tuples: one for each year in the input data. The first field in each tuple is the field being grouped by (the year), and the second field has a bag of tuples for that year. A *bag* is just an unordered collection of tuples, which in Pig Latin is represented using curly braces.

By grouping the data in this way, we have created a row per year, so now all that remains is to find the maximum temperature for the tuples in each bag. Before we do this, let's understand the structure of the grouped_records relation:

```
grunt> DESCRIBE grouped_records;
grouped_records: {group: chararray,filtered_records: {year: chararray,
temperature: int,quality: int}}
```

This tells us that the grouping field is given the alias group by Pig, and the second field is the same structure as the filtered_records relation that was being grouped. With this information, we can try the fourth transformation:

```
grunt> max_temp = FOREACH grouped_records GENERATE group,
>>   MAX(filtered_records.temperature);
```

FOREACH processes every row to generate a derived set of rows, using a GENERATE clause to define the fields in each derived row. In this example, the first field is group, which is just the year. The second field is a little more complex. The filtered_records.temperature reference is to the temperature field of the filtered_records bag in the grouped_records relation. MAX is a built-in function for calculating the maximum value of fields in a bag. In this case, it calculates the maximum temperature for the fields in each filtered_records bag. Let's check the result:

```
grunt> DUMP max_temp;
(1949,111)
(1950,22)
```

We've successfully calculated the maximum temperature for each year.

Generating Examples

In this example, we've used a small sample dataset with just a handful of rows to make it easier to follow the data flow and aid debugging. Creating a cut-down dataset is an art, as ideally it should be rich enough to cover all the cases to exercise your queries (the *completeness* property), yet small enough to make sense to the programmer (the *conciseness* property). Using a random sample doesn't work well in general because join

and filter operations tend to remove all random data, leaving an empty result, which is not illustrative of the general data flow.

With the `ILLUSTRATE` operator, Pig provides a tool for generating a reasonably complete and concise sample dataset. Here is the output from running `ILLUSTRATE` on our dataset (slightly reformatted to fit the page):

```
grunt> ILLUSTRATE max_temp;
-------------------------------------------------------------------------------
| records     | year:chararray      | temperature:int      | quality:int      |
-------------------------------------------------------------------------------
|             | 1949                | 78                   | 1                |
|             | 1949                | 111                  | 1                |
|             | 1949                | 9999                 | 1                |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| filtered_records   | year:chararray    | temperature:int    | quality:int   |
-------------------------------------------------------------------------------
|                    | 1949              | 78                 | 1             |
|                    | 1949              | 111                | 1             |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| grouped_records  | group:chararray   | filtered_records:bag{:tuple(         |
                                          year:chararray,temperature:int,    |
                                          quality:int)}                      |
-------------------------------------------------------------------------------
|                  | 1949              | {(1949, 78, 1), (1949, 111, 1)}      |
-------------------------------------------------------------------------------
-------------------------------------------------
| max_temp     | group:chararray      | :int      |
-------------------------------------------------
|              | 1949                 | 111       |
-------------------------------------------------
```

Notice that Pig used some of the original data (this is important to keep the generated dataset realistic), as well as creating some new data. It noticed the special value 9999 in the query and created a tuple containing this value to exercise the `FILTER` statement.

In summary, the output of `ILLUSTRATE` is easy to follow and can help you understand what your query is doing.

Comparison with Databases

Having seen Pig in action, it might seem that Pig Latin is similar to SQL. The presence of such operators as `GROUP BY` and `DESCRIBE` reinforces this impression. However, there are several differences between the two languages, and between Pig and relational database management systems (RDBMSs) in general.

The most significant difference is that Pig Latin is a data flow programming language, whereas SQL is a declarative programming language. In other words, a Pig Latin pro-

gram is a step-by-step set of operations on an input relation, in which each step is a single transformation. By contrast, SQL statements are a set of constraints that, taken together, define the output. In many ways, programming in Pig Latin is like working at the level of an RDBMS query planner, which figures out how to turn a declarative statement into a system of steps.

RDBMSs store data in tables, with tightly predefined schemas. Pig is more relaxed about the data that it processes: you can define a schema at runtime, but it's optional. Essentially, it will operate on any source of tuples (although the source should support being read in parallel, by being in multiple files, for example), where a UDF is used to read the tuples from their raw representation.[2] The most common representation is a text file with tab-separated fields, and Pig provides a built-in load function for this format. Unlike with a traditional database, there is no data import process to load the data into the RDBMS. The data is loaded from the filesystem (usually HDFS) as the first step in the processing.

Pig's support for complex, nested data structures further differentiates it from SQL, which operates on flatter data structures. Also, Pig's ability to use UDFs and streaming operators that are tightly integrated with the language and Pig's nested data structures makes Pig Latin more customizable than most SQL dialects.

RDBMSs have several features to support online, low-latency queries, such as transactions and indexes, that are absent in Pig. Pig does not support random reads or queries on the order of tens of milliseconds. Nor does it support random writes to update small portions of data; all writes are bulk streaming writes, just like with MapReduce.

Hive (covered in Chapter 17) sits between Pig and conventional RDBMSs. Like Pig, Hive is designed to use HDFS for storage, but otherwise there are some significant differences. Its query language, HiveQL, is based on SQL, and anyone who is familiar with SQL will have little trouble writing queries in HiveQL. Like RDBMSs, Hive mandates that all data be stored in tables, with a schema under its management; however, it can associate a schema with preexisting data in HDFS, so the load step is optional. Pig is able to work with Hive tables using HCatalog; this is discussed further in "Using Hive tables with HCatalog" on page 442.

2. Or as the Pig Philosophy has it, "Pigs eat anything."

Pig Latin

This section gives an informal description of the syntax and semantics of the Pig Latin programming language.[3] It is not meant to offer a complete reference to the language,[4] but there should be enough here for you to get a good understanding of Pig Latin's constructs.

Structure

A Pig Latin program consists of a collection of statements. A statement can be thought of as an operation or a command.[5] For example, a GROUP operation is a type of statement:

```
grouped_records = GROUP records BY year;
```

The command to list the files in a Hadoop filesystem is another example of a statement:

```
ls /
```

Statements are usually terminated with a semicolon, as in the example of the GROUP statement. In fact, this is an example of a statement that must be terminated with a semicolon; it is a syntax error to omit it. The `ls` command, on the other hand, does not have to be terminated with a semicolon. As a general guideline, statements or commands for interactive use in Grunt do not need the terminating semicolon. This group includes the interactive Hadoop commands, as well as the diagnostic operators such as DESCRIBE. It's never an error to add a terminating semicolon, so if in doubt, it's simplest to add one.

Statements that have to be terminated with a semicolon can be split across multiple lines for readability:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
```

Pig Latin has two forms of comments. Double hyphens are used for single-line comments. Everything from the first hyphen to the end of the line is ignored by the Pig Latin interpreter:

```
-- My program
DUMP A; -- What's in A?
```

3. Not to be confused with Pig Latin, the language game. English words are translated into Pig Latin by moving the initial consonant sound to the end of the word and adding an "ay" sound. For example, "pig" becomes "ig-pay," and "Hadoop" becomes "Adoop-hay."
4. Pig Latin does not have a formal language definition as such, but there is a comprehensive guide to the language that you can find through a link on the Pig website.
5. You sometimes see these terms being used interchangeably in documentation on Pig Latin: for example, "GROUP command," "GROUP operation," "GROUP statement."

C-style comments are more flexible since they delimit the beginning and end of the comment block with `/*` and `*/` markers. They can span lines or be embedded in a single line:

```
/*
 * Description of my program spanning
 * multiple lines.
 */
A = LOAD 'input/pig/join/A';
B = LOAD 'input/pig/join/B';
C = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

Pig Latin has a list of keywords that have a special meaning in the language and cannot be used as identifiers. These include the operators (`LOAD`, `ILLUSTRATE`), commands (`cat`, `ls`), expressions (`matches`, `FLATTEN`), and functions (`DIFF`, `MAX`)—all of which are covered in the following sections.

Pig Latin has mixed rules on case sensitivity. Operators and commands are not case sensitive (to make interactive use more forgiving); however, aliases and function names are case sensitive.

Statements

As a Pig Latin program is executed, each statement is parsed in turn. If there are syntax errors or other (semantic) problems, such as undefined aliases, the interpreter will halt and display an error message. The interpreter builds a *logical plan* for every relational operation, which forms the core of a Pig Latin program. The logical plan for the statement is added to the logical plan for the program so far, and then the interpreter moves on to the next statement.

It's important to note that no data processing takes place while the logical plan of the program is being constructed. For example, consider again the Pig Latin program from the first example:

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

When the Pig Latin interpreter sees the first line containing the `LOAD` statement, it confirms that it is syntactically and semantically correct and adds it to the logical plan, but it does *not* load the data from the file (or even check whether the file exists). Indeed, where would it load it? Into memory? Even if it did fit into memory, what would it do

with the data? Perhaps not all the input data is needed (because later statements filter it, for example), so it would be pointless to load it. The point is that it makes no sense to start any processing until the whole flow is defined. Similarly, Pig validates the `GROUP` and `FOREACH...GENERATE` statements, and adds them to the logical plan without executing them. The trigger for Pig to start execution is the `DUMP` statement. At that point, the logical plan is compiled into a physical plan and executed.

Multiquery Execution

Because `DUMP` is a diagnostic tool, it will always trigger execution. However, the `STORE` command is different. In interactive mode, `STORE` acts like `DUMP` and will always trigger execution (this includes the `run` command), but in batch mode it will not (this includes the `exec` command). The reason for this is efficiency. In batch mode, Pig will parse the whole script to see whether there are any optimizations that could be made to limit the amount of data to be written to or read from disk. Consider the following simple example:

```
A = LOAD 'input/pig/multiquery/A';
B = FILTER A BY $1 == 'banana';
C = FILTER A BY $1 != 'banana';
STORE B INTO 'output/b';
STORE C INTO 'output/c';
```

Relations `B` and `C` are both derived from `A`, so to save reading `A` twice, Pig can run this script as a single MapReduce job by reading `A` once and writing two output files from the job, one for each of `B` and `C`. This feature is called *multiquery execution*.

In previous versions of Pig that did not have multiquery execution, each `STORE` statement in a script run in batch mode triggered execution, resulting in a job for each `STORE` statement. It is possible to restore the old behavior by disabling multiquery execution with the `-M` or `-no_multiquery` option to `pig`.

The physical plan that Pig prepares is a series of MapReduce jobs, which in local mode Pig runs in the local JVM and in MapReduce mode Pig runs on a Hadoop cluster.

You can see the logical and physical plans created by Pig using the `EXPLAIN` command on a relation (`EXPLAIN max_temp;`, for example).

`EXPLAIN` will also show the MapReduce plan, which shows how the physical operators are grouped into MapReduce jobs. This is a good way to find out how many MapReduce jobs Pig will run for your query.

The relational operators that can be a part of a logical plan in Pig are summarized in Table 16-1. We go through the operators in more detail in "Data Processing Operators" on page 456.

Table 16-1. Pig Latin relational operators

Category	Operator	Description
Loading and storing	`LOAD`	Loads data from the filesystem or other storage into a relation
	`STORE`	Saves a relation to the filesystem or other storage
	`DUMP` (`\d`)	Prints a relation to the console
Filtering	`FILTER`	Removes unwanted rows from a relation
	`DISTINCT`	Removes duplicate rows from a relation
	`FOREACH...GENERATE`	Adds or removes fields to or from a relation
	`MAPREDUCE`	Runs a MapReduce job using a relation as input
	`STREAM`	Transforms a relation using an external program
	`SAMPLE`	Selects a random sample of a relation
	`ASSERT`	Ensures a condition is true for all rows in a relation; otherwise, fails
Grouping and joining	`JOIN`	Joins two or more relations
	`COGROUP`	Groups the data in two or more relations
	`GROUP`	Groups the data in a single relation
	`CROSS`	Creates the cross product of two or more relations
	`CUBE`	Creates aggregations for all combinations of specified columns in a relation
Sorting	`ORDER`	Sorts a relation by one or more fields
	`RANK`	Assign a rank to each tuple in a relation, optionally sorting by fields first
	`LIMIT`	Limits the size of a relation to a maximum number of tuples
Combining and splitting	`UNION`	Combines two or more relations into one
	`SPLIT`	Splits a relation into two or more relations

There are other types of statements that are not added to the logical plan. For example, the diagnostic operators—`DESCRIBE`, `EXPLAIN`, and `ILLUSTRATE`—are provided to allow the user to interact with the logical plan for debugging purposes (see Table 16-2). `DUMP` is a sort of diagnostic operator, too, since it is used only to allow interactive debugging of small result sets or in combination with `LIMIT` to retrieve a few rows from a larger relation. The `STORE` statement should be used when the size of the output is more than a few lines, as it writes to a file rather than to the console.

Table 16-2. Pig Latin diagnostic operators

Operator (Shortcut)	Description
`DESCRIBE` (`\de`)	Prints a relation's schema
`EXPLAIN` (`\e`)	Prints the logical and physical plans
`ILLUSTRATE` (`\i`)	Shows a sample execution of the logical plan, using a generated subset of the input

Pig Latin also provides three statements—`REGISTER`, `DEFINE`, and `IMPORT`—that make it possible to incorporate macros and user-defined functions into Pig scripts (see Table 16-3).

Table 16-3. Pig Latin macro and UDF statements

Statement	Description
`REGISTER`	Registers a JAR file with the Pig runtime
`DEFINE`	Creates an alias for a macro, UDF, streaming script, or command specification
`IMPORT`	Imports macros defined in a separate file into a script

Because they do not process relations, commands are not added to the logical plan; instead, they are executed immediately. Pig provides commands to interact with Hadoop filesystems (which are very handy for moving data around before or after processing with Pig) and MapReduce, as well as a few utility commands (described in Table 16-4).

Table 16-4. Pig Latin commands

Category	Command	Description
Hadoop filesystem	`cat`	Prints the contents of one or more files
	`cd`	Changes the current directory
	`copyFromLocal`	Copies a local file or directory to a Hadoop filesystem
	`copyToLocal`	Copies a file or directory on a Hadoop filesystem to the local filesystem
	`cp`	Copies a file or directory to another directory
	`fs`	Accesses Hadoop's filesystem shell
	`ls`	Lists files
	`mkdir`	Creates a new directory
	`mv`	Moves a file or directory to another directory
	`pwd`	Prints the path of the current working directory
	`rm`	Deletes a file or directory
	`rmf`	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)
Hadoop MapReduce	`kill`	Kills a MapReduce job

Category	Command	Description
Utility	`clear`	Clears the screen in Grunt
	`exec`	Runs a script in a new Grunt shell in batch mode
	`help`	Shows the available commands and options
	`history`	Prints the query statements run in the current Grunt session
	`quit` (`\q`)	Exits the interpreter
	`run`	Runs a script within the existing Grunt shell
	`set`	Sets Pig options and MapReduce job properties
	`sh`	Runs a shell command from within Grunt

The filesystem commands can operate on files or directories in any Hadoop filesystem, and they are very similar to the `hadoop fs` commands (which is not surprising, as both are simple wrappers around the Hadoop `FileSystem` interface). You can access all of the Hadoop filesystem shell commands using Pig's `fs` command. For example, `fs -ls` will show a file listing, and `fs -help` will show help on all the available commands.

Precisely which Hadoop filesystem is used is determined by the `fs.defaultFS` property in the site file for Hadoop Core. See "The Command-Line Interface" on page 50 for more details on how to configure this property.

These commands are mostly self-explanatory, except `set`, which is used to set options that control Pig's behavior (including arbitrary MapReduce job properties). The `debug` option is used to turn debug logging on or off from within a script (you can also control the log level when launching Pig, using the `-d` or `-debug` option):

```
grunt> set debug on
```

Another useful option is the `job.name` option, which gives a Pig job a meaningful name, making it easier to pick out your Pig MapReduce jobs when running on a shared Hadoop cluster. If Pig is running a script (rather than operating as an interactive query from Grunt), its job name defaults to a value based on the script name.

There are two commands in Table 16-4 for running a Pig script, `exec` and `run`. The difference is that `exec` runs the script in batch mode in a new Grunt shell, so any aliases defined in the script are not accessible to the shell after the script has completed. On the other hand, when running a script with `run`, it is as if the contents of the script had been entered manually, so the command history of the invoking shell contains all the statements from the script. Multiquery execution, where Pig executes a batch of statements in one go (see "Multiquery Execution" on page 434), is used only by `exec`, not `run`.

Control Flow

By design, Pig Latin lacks native control flow statements. The recommended approach for writing programs that have conditional logic or loop constructs is to embed Pig Latin in another language, such as Python, JavaScript, or Java, and manage the control flow from there. In this model, the host script uses a compile-bind-run API to execute Pig scripts and retrieve their status. Consult the Pig documentation for details of the API.

Embedded Pig programs always run in a JVM, so for Python and JavaScript you use the `pig` command followed by the name of your script, and the appropriate Java scripting engine will be selected (Jython for Python, Rhino for JavaScript).

Expressions

An expression is something that is evaluated to yield a value. Expressions can be used in Pig as a part of a statement containing a relational operator. Pig has a rich variety of expressions, many of which will be familiar from other programming languages. They are listed in Table 16-5, with brief descriptions and examples. We will see examples of many of these expressions throughout the chapter.

Table 16-5. Pig Latin expressions

Category	Expressions	Description	Examples
Constant	Literal	Constant value (see also the "Literal example" column in Table 16-6)	`1.0`, `'a'`
Field (by position)	`$`*n*	Field in position *n* (zero-based)	`$0`
Field (by name)	*f*	Field named *f*	`year`
Field (disambiguate)	*r*`::`*f*	Field named *f* from relation *r* after grouping or joining	`A::year`
Projection	*c*`.$`*n*, *c*`.`*f*	Field in container *c* (relation, bag, or tuple) by position, by name	`records.$0`, `records.year`
Map lookup	*m*`#`*k*	Value associated with key *k* in map *m*	`items#'Coat'`
Cast	`(`*t*`)` *f*	Cast of field *f* to type *t*	`(int) year`
Arithmetic	*x* `+` *y*, *x* `-` *y*	Addition, subtraction	`$1 + $2`, `$1 - $2`
	x `*` *y*, *x* `/` *y*	Multiplication, division	`$1 * $2`, `$1 / $2`
	x `%` *y*	Modulo, the remainder of *x* divided by *y*	`$1 % $2`
	`+`*x*, `-`*x*	Unary positive, negation	`+1`, `-1`
Conditional	*x* `?` *y* `:` *z*	Bincond/ternary; *y* if *x* evaluates to true, *z* otherwise	`quality == 0 ? 0 : 1`
	`CASE`	Multi-case conditional	`CASE q WHEN 0 THEN 'good' ELSE 'bad' END`

Category	Expressions	Description	Examples
Comparison	`x == y`, `x != y`	Equals, does not equal	`quality == 0`, `tempera ture != 9999`
	`x > y`, `x < y`	Greater than, less than	`quality > 0`, `quality < 10`
	`x >= y`, `x <= y`	Greater than or equal to, less than or equal to	`quality >= 1`, `quality <= 9`
	`x matches y`	Pattern matching with regular expression	`quality matches '[01459]'`
	`x is null`	Is null	`temperature is null`
	`x is not null`	Is not null	`temperature is not null`
Boolean	`x OR y`	Logical OR	`q == 0 OR q == 1`
	`x AND y`	Logical AND	`q == 0 AND r == 0`
	`NOT x`	Logical negation	`NOT q matches '[01459]'`
	`IN x`	Set membership	`q IN (0, 1, 4, 5, 9)`
Functional	*fn*(*f1*,*f2*,...)	Invocation of function *fn* on fields *f1*, *f2*, etc.	`isGood(quality)`
Flatten	`FLATTEN(`*f*`)`	Removal of a level of nesting from bags and tuples	`FLATTEN(group)`

Types

So far you have seen some of the simple types in Pig, such as `int` and `chararray`. Here we will discuss Pig's built-in types in more detail.

Pig has a `boolean` type and six numeric types: `int`, `long`, `float`, `double`, `biginteger`, and `bigdecimal`, which are identical to their Java counterparts. There is also a `bytearray` type, like Java's `byte` array type for representing a blob of binary data, and `chararray`, which, like `java.lang.String`, represents textual data in UTF-16 format (although it can be loaded or stored in UTF-8 format). The `datetime` type is for storing a date and time with millisecond precision and including a time zone.

Pig does not have types corresponding to Java's `byte`, `short`, or `char` primitive types. These are all easily represented using Pig's `int` type, or `chararray` for `char`.

The Boolean, numeric, textual, binary, and temporal types are simple atomic types. Pig Latin also has three complex types for representing nested structures: `tuple`, `bag`, and `map`. All of Pig Latin's types are listed in Table 16-6.

Table 16-6. Pig Latin types

Category	Type	Description	Literal example
Boolean	`boolean`	True/false value	`true`
Numeric	`int`	32-bit signed integer	`1`
	`long`	64-bit signed integer	`1L`
	`float`	32-bit floating-point number	`1.0F`
	`double`	64-bit floating-point number	`1.0`
	`biginteger`	Arbitrary-precision integer	`'10000000000'`
	`bigdecimal`	Arbitrary-precision signed decimal number	`'0.11000100000000000000000001'`
Text	`chararray`	Character array in UTF-16 format	`'a'`
Binary	`bytearray`	Byte array	Not supported
Temporal	`datetime`	Date and time with time zone	Not supported, use `ToDate` built-in function
Complex	`tuple`	Sequence of fields of any type	`(1,'pomegranate')`
	`bag`	Unordered collection of tuples, possibly with duplicates	`{(1,'pomegranate'),(2)}`
	`map`	Set of key-value pairs; keys must be character arrays, but values may be any type	`['a'#'pomegranate']`

The complex types are usually loaded from files or constructed using relational operators. Be aware, however, that the literal form in Table 16-6 is used when a constant value is created from within a Pig Latin program. The raw form in a file is usually different when using the standard `PigStorage` loader. For example, the representation in a file of the bag in Table 16-6 would be `{(1,pomegranate),(2)}` (note the lack of quotation marks), and with a suitable schema, this would be loaded as a relation with a single field and row, whose value was the bag.

Pig provides the built-in functions `TOTUPLE`, `TOBAG`, and `TOMAP`, which are used for turning expressions into tuples, bags, and maps.

Although relations and bags are conceptually the same (unordered collections of tuples), in practice Pig treats them slightly differently. A relation is a top-level construct, whereas a bag has to be contained in a relation. Normally you don't have to worry about this, but there are a few restrictions that can trip up the uninitiated. For example, it's not possible to create a relation from a bag literal. So, the following statement fails:

```
A = {(1,2),(3,4)}; -- Error
```

The simplest workaround in this case is to load the data from a file using the `LOAD` statement.

As another example, you can't treat a relation like a bag and project a field into a new relation (`$0` refers to the first field of `A`, using the positional notation):

```
B = A.$0;
```

Instead, you have to use a relational operator to turn the relation A into relation B:

```
B = FOREACH A GENERATE $0;
```

It's possible that a future version of Pig Latin will remove these inconsistencies and treat relations and bags in the same way.

Schemas

A relation in Pig may have an associated schema, which gives the fields in the relation names and types. We've seen how an AS clause in a LOAD statement is used to attach a schema to a relation:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year:int, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: int,temperature: int,quality: int}
```

This time we've declared the year to be an integer rather than a chararray, even though the file it is being loaded from is the same. An integer may be more appropriate if we need to manipulate the year arithmetically (to turn it into a timestamp, for example), whereas the chararray representation might be more appropriate when it's being used as a simple identifier. Pig's flexibility in the degree to which schemas are declared contrasts with schemas in traditional SQL databases, which are declared before the data is loaded into the system. Pig is designed for analyzing plain input files with no associated type information, so it is quite natural to choose types for fields later than you would with an RDBMS.

It's possible to omit type declarations completely, too:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature, quality);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

In this case, we have specified only the names of the fields in the schema: year, temperature, and quality. The types default to bytearray, the most general type, representing a binary string.

You don't need to specify types for every field; you can leave some to default to bytearray, as we have done for year in this declaration:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>>   AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

However, if you specify a schema in this way, you do need to specify every field. Also, there's no way to specify the type of a field without specifying the name. On the other hand, the schema is entirely optional and can be omitted by not specifying an AS clause:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

Fields in a relation with no schema can be referenced using only positional notation: `$0` refers to the first field in a relation, `$1` to the second, and so on. Their types default to `bytearray`:

```
grunt> projected_records = FOREACH records GENERATE $0, $1, $2;
grunt> DUMP projected_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

Although it can be convenient not to assign types to fields (particularly in the first stages of writing a query), doing so can improve the clarity and efficiency of Pig Latin programs and is generally recommended.

Using Hive tables with HCatalog

Declaring a schema as a part of the query is flexible but doesn't lend itself to schema reuse. A set of Pig queries over the same input data will often have the same schema repeated in each query. If the query processes a large number of fields, this repetition can become hard to maintain.

HCatalog (which is a component of Hive) solves this problem by providing access to Hive's metastore, so that Pig queries can reference schemas by name, rather than specifying them in full each time. For example, after running through "An Example" on page 474 to load data into a Hive table called `records`, Pig can access the table's schema and data as follows:

```
% pig -useHCatalog
grunt> records = LOAD 'records' USING org.apache.hcatalog.pig.HCatLoader();
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

Validation and nulls

A SQL database will enforce the constraints in a table's schema at load time; for example, trying to load a string into a column that is declared to be a numeric type will fail. In

Pig, if the value cannot be cast to the type declared in the schema, it will substitute a `null` value. Let's see how this works when we have the following input for the weather data, which has an "e" character in place of an integer:

```
1950 0   1
1950 22  1
1950 e   1
1949 111 1
1949 78  1
```

Pig handles the corrupt line by producing a `null` for the offending value, which is displayed as the absence of a value when dumped to screen (and also when saved using `STORE`):

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>>   AS (year:chararray, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,,1)
(1949,111,1)
(1949,78,1)
```

Pig produces a warning for the invalid field (not shown here) but does not halt its processing. For large datasets, it is very common to have corrupt, invalid, or merely unexpected data, and it is generally infeasible to incrementally fix every unparsable record. Instead, we can pull out all of the invalid records in one go so we can take action on them, perhaps by fixing our program (because they indicate that we have made a mistake) or by filtering them out (because the data is genuinely unusable):

```
grunt> corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
```

Note the use of the `is null` operator, which is analogous to SQL. In practice, we would include more information from the original record, such as an identifier and the value that could not be parsed, to help our analysis of the bad data.

We can find the number of corrupt records using the following idiom for counting the number of rows in a relation:

```
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group, COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1)
```

("GROUP" on page 464 explains grouping and the `ALL` operation in more detail.)

Another useful technique is to use the `SPLIT` operator to partition the data into "good" and "bad" relations, which can then be analyzed separately:

```
grunt> SPLIT records INTO good_records IF temperature is not null,
>>   bad_records OTHERWISE;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

Going back to the case in which `temperature`'s type was left undeclared, the corrupt data cannot be detected easily, since it doesn't surface as a `null`:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>>   AS (year:chararray, temperature, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,e,1)
(1949,111,1)
(1949,78,1)
grunt> filtered_records = FILTER records BY temperature != 9999 AND
>>   quality IN (0, 1, 4, 5, 9);
grunt> grouped_records = GROUP filtered_records BY year;
grunt> max_temp = FOREACH grouped_records GENERATE group,
>>   MAX(filtered_records.temperature);
grunt> DUMP max_temp;
(1949,111.0)
(1950,22.0)
```

What happens in this case is that the `temperature` field is interpreted as a `bytearray`, so the corrupt field is not detected when the input is loaded. When passed to the `MAX` function, the `temperature` field is cast to a `double`, since `MAX` works only with numeric types. The corrupt field cannot be represented as a `double`, so it becomes a `null`, which `MAX` silently ignores. The best approach is generally to declare types for your data on loading and look for missing or corrupt values in the relations themselves before you do your main processing.

Sometimes corrupt data shows up as smaller tuples because fields are simply missing. You can filter these out by using the `SIZE` function as follows:

```
grunt> A = LOAD 'input/pig/corrupt/missing_fields';
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3)
(1,Scarf)
grunt> B = FILTER A BY SIZE(TOTUPLE(*)) > 1;
grunt> DUMP B;
(2,Tie)
(4,Coat)
(1,Scarf)
```

The STREAM operator uses PigStorage to serialize and deserialize relations to and from the program's standard input and output streams. Tuples in A are converted to tab-delimited lines that are passed to the script. The output of the script is read one line at a time and split on tabs to create new tuples for the output relation C. You can provide a custom serializer and deserializer by subclassing PigStreamingBase (in the org.apache.pig package), then using the DEFINE operator.

Pig streaming is most powerful when you write custom processing scripts. The following Python script filters out bad weather records:

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

import re
import sys

for line in sys.stdin:
  (year, temp, q) = line.strip().split()
  if (temp != "9999" and re.match("[01459]", q)):
    print "%s\t%s" % (year, temp)
```

To use the script, you need to ship it to the cluster. This is achieved via a DEFINE clause, which also creates an alias for the STREAM command. The STREAM statement can then refer to the alias, as the following Pig script shows:

```
-- max_temp_filter_stream.pig
DEFINE is_good_quality `is_good_quality.py`
  SHIP ('ch16-pig/src/main/python/is_good_quality.py');
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = STREAM records THROUGH is_good_quality
  AS (year:chararray, temperature:int);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

Grouping and Joining Data

Joining datasets in MapReduce takes some work on the part of the programmer (see "Joins" on page 268), whereas Pig has very good built-in support for join operations, making it much more approachable. Since the large datasets that are suitable for analysis by Pig (and MapReduce in general) are not normalized, however, joins are used more infrequently in Pig than they are in SQL.

JOIN

Let's look at an example of an inner join. Consider the relations A and B:

```
grunt> DUMP A;
(2,Tie)
```

```
(4,Coat)
(3,Hat)
(1,Scarf)
grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

We can join the two relations on the numerical (identity) field in each:

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

This is a classic inner join, where each match between the two relations corresponds to a row in the result. (It's actually an equijoin because the join predicate is equality.) The result's fields are made up of all the fields of all the input relations.

You should use the general join operator when all the relations being joined are too large to fit in memory. If one of the relations is small enough to fit in memory, you can use a special type of join called a *fragment replicate join*, which is implemented by distributing the small input to all the mappers and performing a map-side join using an in-memory lookup table against the (fragmented) larger relation. There is a special syntax for telling Pig to use a fragment replicate join:[9]

```
grunt> C = JOIN A BY $0, B BY $1 USING 'replicated';
```

The first relation must be the large one, followed by one or more small ones (all of which must fit in memory).

Pig also supports outer joins using a syntax that is similar to SQL's (this is covered for Hive in "Outer joins" on page 506). For example:

```
grunt> C = JOIN A BY $0 LEFT OUTER, B BY $1;
grunt> DUMP C;
(1,Scarf,,)
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

9. There are more keywords that may be used in the `USING` clause, including `'skewed'` (for large datasets with a skewed keyspace), `'merge'` (to effect a merge join for inputs that are already sorted on the join key), and `'merge-sparse'` (where 1% or less of data is matched). See Pig's documentation for details on how to use these specialized joins.

COGROUP

foealways gives a flat structure: a set of tuples. The fB2f statement is similar to foe, but instead creates a nested set of output tuples. This can be useful if you want to exploit the structure in subsequent statements:

```
;>CcP(JD = COGROUP A BY $0, B BY $1;
;>CcP(JDUMP D;
3rE   E  3TRUErv  v
3,E  3,E {G>Lv  E   v
3  E  3  EkUuv  E  3  Gc  E  vE3  auE  v  v
3  E  3  E  GPv  E  3$ uE  v  v
3)E  3)E  aGPv  E  3  Gc  E)v  v
```

fB2f generates a tuple for each unique grouping key. The first field of each tuple is the key, and the remaining fields are bags of tuples from the relations with a matching key. The first bag contains the matching tuples from relation T with the same key. Similarly, the second bag contains the matching tuples from relation with the same key.

If for a particular key a relation has no matching key, the bag for that relation is empty. For example, since no one has bought a scarf (with ID 1), the second bag in the tuple for that row is empty. This is an example of an outer join, which is the default type for fB2f . It can be made explicit using the f k$2keyword, making this fB2f statement the same as the previous one:

```
HJOJ  fB2f   JTJ   JOUTEREJ  J   J QUTER
```

You can suppress rows with empty bags by using the oee$2 keyword, which gives the fB2f inner join semantics. The oee$2 keyword is applied per relation, so the following suppresses rows only when relation T has no match (dropping the unknown product 0 here):

```
;>CcP(JE = COGROUP A BY $0 INNER, B BY $1;
;>CcP(JDUMP E;
3,E  3,E {G>Lv  E   v
3  E  3  EkUuv  E  3  Gc  E  vE3  auE  v  v
3  E  3  E  GPv  E  3$ uE  v  v
3)E  3)E  aGPv  E  3  Gc  E)v  v
```

We can flatten this structure to discover who bought each of the items in relation T:

```
;>CcP(JF = FOREACH E GENERATE FLATTEN(A), B.$0;
;>CcP(JDUMP F;
3,E {G>LE   v
3  EkUuE  3  Gc  vE3  auv  v
3  E  GPE  3$ uv  v
3)E  aGPE  3  Gc  v  v
```

Using a combination of fB2f , oee$2, and =YTkk$e (which removes nesting) it's possible to simulate an (inner) foe

```
grunt> G = COGROUP A BY $0 INNER, B BY $1 INNER;
grunt> H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);
grunt> DUMP H;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

This gives the same result as `JOIN A BY $0, B BY $1`.

If the join key is composed of several fields, you can specify them all in the `BY` clauses of the `JOIN` or `COGROUP` statement. Make sure that the number of fields in each `BY` clause is the same.

Here's another example of a join in Pig, in a script for calculating the maximum temperature for every station over a time period controlled by the input:

```
-- max_temp_station_name.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();

stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt'
  USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')
  AS (usaf:chararray, wban:chararray, name:chararray);

trimmed_stations = FOREACH stations GENERATE usaf, wban, TRIM(name);

records = LOAD 'input/ncdc/all/191*'
  USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93')
  AS (usaf:chararray, wban:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30;
max_temp = FOREACH grouped_records GENERATE FLATTEN(group),
  MAX(filtered_records.temperature);
max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf, wban)
  PARALLEL 30;
max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2;

STORE max_temp_result INTO 'max_temp_by_station';
```

We use the cut UDF we developed earlier to load one relation holding the station IDs (USAF and WBAN identifiers) and names, and one relation holding all the weather records, keyed by station ID. We group the filtered weather records by station ID and aggregate by maximum temperature before joining with the stations. Finally, we project out the fields we want in the final result: USAF, WBAN, station name, and maximum temperature.

Here are a few results for the 1910s:

```
228020        99999        SORTAVALA      322
029110        99999        VAASA AIRPORT  300
040650        99999        GRIMSEY        378
```

This query could be made more efficient by using a fragment replicate join, as the station metadata is small.

CROSS

Pig Latin includes the cross-product operator (also known as the Cartesian product), `CROSS`, which joins every tuple in a relation with every tuple in a second relation (and with every tuple in further relations, if supplied). The size of the output is the product of the size of the inputs, potentially making the output very large:

```
grunt> I = CROSS A, B;
grunt> DUMP I;
(2,Tie,Joe,2)
(2,Tie,Hank,4)
(2,Tie,Ali,0)
(2,Tie,Eve,3)
(2,Tie,Hank,2)
(4,Coat,Joe,2)
(4,Coat,Hank,4)
(4,Coat,Ali,0)
(4,Coat,Eve,3)
(4,Coat,Hank,2)
(3,Hat,Joe,2)
(3,Hat,Hank,4)
(3,Hat,Ali,0)
(3,Hat,Eve,3)
(3,Hat,Hank,2)
(1,Scarf,Joe,2)
(1,Scarf,Hank,4)
(1,Scarf,Ali,0)
(1,Scarf,Eve,3)
(1,Scarf,Hank,2)
```

When dealing with large datasets, you should try to avoid operations that generate intermediate representations that are quadratic (or worse) in size. Computing the cross product of the whole input dataset is rarely needed, if ever.

For example, at first blush, one might expect that calculating pairwise document similarity in a corpus of documents would require every document pair to be generated before calculating their similarity. However, if we start with the insight that most document pairs have a similarity score of zero (i.e., they are unrelated), then we can find a way to a better algorithm.

In this case, the key idea is to focus on the entities that we are using to calculate similarity (terms in a document, for example) and make them the center of the algorithm. In practice, we also remove terms that don't help discriminate between documents (stop-

words), and this reduces the problem space still further. Using this technique to analyze a set of roughly one million (10^6) documents generates on the order of one billion (10^9) intermediate pairs,[10] rather than the one trillion (10^{12}) produced by the naive approach (generating the cross product of the input) or the approach with no stopword removal.

GROUP

Where `COGROUP` groups the data in two or more relations, the `GROUP` statement groups the data in a single relation. `GROUP` supports grouping by more than equality of keys: you can use an expression or user-defined function as the group key. For example, consider the following relation `A`:

```
grunt> DUMP A;
(Joe,cherry)
(Ali,apple)
(Joe,banana)
(Eve,apple)
```

Let's group by the number of characters in the second field:

```
grunt> B = GROUP A BY SIZE($1);
grunt> DUMP B;
(5,{(Eve,apple),(Ali,apple)})
(6,{(Joe,banana),(Joe,cherry)})
```

`GROUP` creates a relation whose first field is the grouping field, which is given the alias `group`. The second field is a bag containing the grouped fields with the same schema as the original relation (in this case, `A`).

There are also two special grouping operations: `ALL` and `ANY`. `ALL` groups all the tuples in a relation in a single group, as if the `GROUP` function were a constant:

```
grunt> C = GROUP A ALL;
grunt> DUMP C;
(all,{(Eve,apple),(Joe,banana),(Ali,apple),(Joe,cherry)})
```

Note that there is no `BY` in this form of the `GROUP` statement. The `ALL` grouping is commonly used to count the number of tuples in a relation, as shown in "Validation and nulls" on page 442.

The `ANY` keyword is used to group the tuples in a relation randomly, which can be useful for sampling.

10. Tamer Elsayed, Jimmy Lin, and Douglas W. Oard, "Pairwise Document Similarity in Large Collections with MapReduce," *Proceedings of the 46th Annual Meeting of the Association of Computational Linguistics*, June 2008.

Sorting Data

Relations are unordered in Pig. Consider a relation A:

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
```

There is no guarantee which order the rows will be processed in. In particular, when retrieving the contents of A using DUMP or STORE, the rows may be written in any order. If you want to impose an order on the output, you can use the ORDER operator to sort a relation by one or more fields. The default sort order compares fields of the same type using the natural ordering, and different types are given an arbitrary, but deterministic, ordering (a tuple is always "less than" a bag, for example).

The following example sorts A by the first field in ascending order and by the second field in descending order:

```
grunt> B = ORDER A BY $0, $1 DESC;
grunt> DUMP B;
(1,2)
(2,4)
(2,3)
```

Any further processing on a sorted relation is not guaranteed to retain its order. For example:

```
grunt> C = FOREACH B GENERATE *;
```

Even though relation C has the same contents as relation B, its tuples may be emitted in any order by a DUMP or a STORE. It is for this reason that it is usual to perform the ORDER operation just before retrieving the output.

The LIMIT statement is useful for limiting the number of results as a quick-and-dirty way to get a sample of a relation. (Although random sampling using the SAMPLE operator, or prototyping with the ILLUSTRATE command, should be preferred for generating more representative samples of the data.) It can be used immediately after the ORDER statement to retrieve the first *n* tuples. Usually, LIMIT will select any *n* tuples from a relation, but when used immediately after an ORDER statement, the order is retained (in an exception to the rule that processing a relation does not retain its order):

```
grunt> D = LIMIT B 2;
grunt> DUMP D;
(1,2)
(2,4)
```

If the limit is greater than the number of tuples in the relation, all tuples are returned (so LIMIT has no effect).

Using LIMIT can improve the performance of a query because Pig tries to apply the limit as early as possible in the processing pipeline, to minimize the amount of data that needs to be processed. For this reason, you should always use LIMIT if you are not interested in the entire output.

Combining and Splitting Data

Sometimes you have several relations that you would like to combine into one. For this, the UNION statement is used. For example:

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
grunt> DUMP B;
(z,x,8)
(w,y,1)
grunt> C = UNION A, B;
grunt> DUMP C;
(2,3)
(z,x,8)
(1,2)
(w,y,1)
(2,4)
```

C is the union of relations A and B, and because relations are unordered, the order of the tuples in C is undefined. Also, it's possible to form the union of two relations with different schemas or with different numbers of fields, as we have done here. Pig attempts to merge the schemas from the relations that UNION is operating on. In this case, they are incompatible, so C has no schema:

```
grunt> DESCRIBE A;
A: {f0: int,f1: int}
grunt> DESCRIBE B;
B: {f0: chararray,f1: chararray,f2: int}
grunt> DESCRIBE C;
Schema for C unknown.
```

If the output relation has no schema, your script needs to be able to handle tuples that vary in the number of fields and/or types.

The SPLIT operator is the opposite of UNION: it partitions a relation into two or more relations. See "Validation and nulls" on page 442 for an example of how to use it.

Pig in Practice

There are some practical techniques that are worth knowing about when you are developing and running Pig programs. This section covers some of them.

to set logging configuration for the session. For example, the following handy invocation will send debug messages to the console:

```
% hive -hiveconf hive.root.logger=DEBUG,console
```

Hive Services

The Hive shell is only one of several services that you can run using the `hive` command. You can specify the service to run using the `--service` option. Type `hive --service help` to get a list of available service names; some of the most useful ones are described in the following list:

`cli`
: The command-line interface to Hive (the shell). This is the default service.

`hiveserver2`
: Runs Hive as a server exposing a Thrift service, enabling access from a range of clients written in different languages. HiveServer 2 improves on the original HiveServer by supporting authentication and multiuser concurrency. Applications using the Thrift, JDBC, and ODBC connectors need to run a Hive server to communicate with Hive. Set the `hive.server2.thrift.port` configuration property to specify the port the server will listen on (defaults to 10000).

`beeline`
: A command-line interface to Hive that works in embedded mode (like the regular CLI), or by connecting to a HiveServer 2 process using JDBC.

`hwi`
: The Hive Web Interface. A simple web interface that can be used as an alternative to the CLI without having to install any client software. See also Hue for a more fully featured Hadoop web interface that includes applications for running Hive queries and browsing the Hive metastore.

`jar`
: The Hive equivalent of `hadoop jar`, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.

`metastore`
: By default, the metastore is run in the same process as the Hive service. Using this service, it is possible to run the metastore as a standalone (remote) process. Set the `METASTORE_PORT` environment variable (or use the `-p` command-line option) to specify the port the server will listen on (defaults to 9083).

Hive clients

If you run Hive as a server (`hive --service hiveserver2`), there are a number of different mechanisms for connecting to it from applications (the relationship between Hive clients and Hive services is illustrated in Figure 17-1):

Thrift Client
: The Hive server is exposed as a Thrift service, so it's possible to interact with it using any programming language that supports Thrift. There are third-party projects providing clients for Python and Ruby; for more details, see the Hive wiki.

JDBC driver
: Hive provides a Type 4 (pure Java) JDBC driver, defined in the class `org.apache.hadoop.hive.jdbc.HiveDriver`. When configured with a JDBC URI of the form `jdbc:hive2://host:port/dbname`, a Java application will connect to a Hive server running in a separate process at the given host and port. (The driver makes calls to an interface implemented by the Hive Thrift Client using the Java Thrift bindings.)

  You may alternatively choose to connect to Hive via JDBC in *embedded mode* using the URI `jdbc:hive2://`. In this mode, Hive runs in the same JVM as the application invoking it; there is no need to launch it as a standalone server, since it does not use the Thrift service or the Hive Thrift Client.

  The Beeline CLI uses the JDBC driver to communicate with Hive.

ODBC driver
: An ODBC driver allows applications that support the ODBC protocol (such as business intelligence software) to connect to Hive. The Apache Hive distribution does not ship with an ODBC driver, but several vendors make one freely available. (Like the JDBC driver, ODBC drivers use Thrift to communicate with the Hive server.)

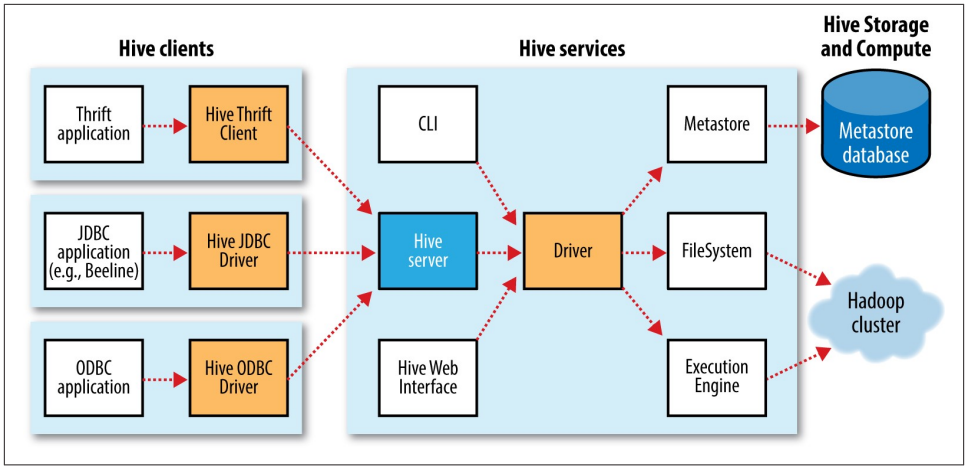

Figure 17-1. Hive architecture

The Metastore

The *metastore* is the central repository of Hive metadata. The metastore is divided into two pieces: a service and the backing store for the data. By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk. This is called the *embedded metastore* configuration (see Figure 17-2).

Using an embedded metastore is a simple way to get started with Hive; however, only one embedded Derby database can access the database files on disk at any one time, which means you can have only one Hive session open at a time that accesses the same metastore. Trying to start a second session produces an error when it attempts to open a connection to the metastore.

The solution to supporting multiple sessions (and therefore multiple users) is to use a standalone database. This configuration is referred to as a *local metastore*, since the metastore service still runs in the same process as the Hive service but connects to a database running in a separate process, either on the same machine or on a remote machine. Any JDBC-compliant database may be used by setting the `javax.jdo.option.*` configuration properties listed in Table 17-1.[3]

3. The properties have the `javax.jdo` prefix because the metastore implementation uses the Java Data Objects (JDO) API for persisting Java objects. Specifically, it uses the DataNucleus implementation of JDO.

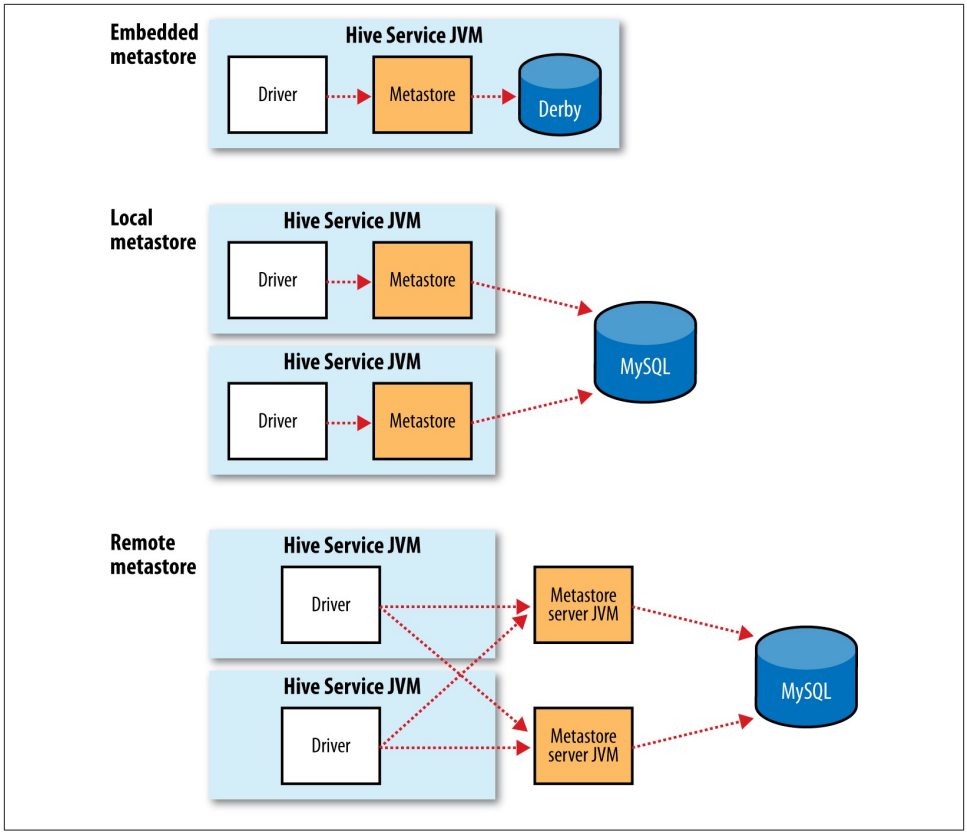

Figure 17-2. Metastore configurations

MySQL is a popular choice for the standalone metastore. In this case, the `javax.jdo.option.ConnectionURL` property is set to `jdbc:mysql://host/dbname?createDatabaseIfNotExist=true`, and `javax.jdo.option.ConnectionDriverName` is set to `com.mysql.jdbc.Driver`. (The username and password should be set too, of course.) The JDBC driver JAR file for MySQL (Connector/J) must be on Hive's classpath, which is simply achieved by placing it in Hive's *lib* directory.

Going a step further, there's another metastore configuration called a *remote metastore*, where one or more metastore servers run in separate processes to the Hive service. This brings better manageability and security because the database tier can be completely firewalled off, and the clients no longer need the database credentials.

A Hive service is configured to use a remote metastore by setting `hive.metastore.uris` to the metastore server URI(s), separated by commas if there is more than one. Metastore server URIs are of the form `thrift://host:port`, where the port

corresponds to the one set by `METASTORE_PORT` when starting the metastore server (see "Hive Services" on page 478).

Table 17-1. Important metastore configuration properties

Property name	Type	Default value	Description
`hive.metastore.warehouse.dir`	URI	`/user/hive/warehouse`	The directory relative to `fs.defaultFS` where managed tables are stored.
`hive.metastore.uris`	Comma-separated URIs	Not set	If not set (the default), use an in-process metastore; otherwise, connect to one or more remote metastores, specified by a list of URIs. Clients connect in a round-robin fashion when there are multiple remote servers.
`javax.jdo.option.ConnectionURL`	URI	`jdbc:derby:;databaseName=metastore_db;create=true`	The JDBC URL of the metastore database.
`javax.jdo.option.ConnectionDriverName`	`String`	`org.apache.derby.jdbc.EmbeddedDriver`	The JDBC driver classname.
`javax.jdo.option.ConnectionUserName`	`String`	`APP`	The JDBC username.
`javax.jdo.option.ConnectionPassword`	`String`	`mine`	The JDBC password.

Comparison with Traditional Databases

Although Hive resembles a traditional database in many ways (such as supporting a SQL interface), its original HDFS and MapReduce underpinnings mean that there are a number of architectural differences that have directly influenced the features that Hive supports. Over time, however, these limitations have been (and continue to be) removed, with the result that Hive looks and feels more like a traditional database with every year that passes.

Schema on Read Versus Schema on Write

In a traditional database, a table's schema is enforced at data load time. If the data being loaded doesn't conform to the schema, then it is rejected. This design is sometimes called *schema on write* because the data is checked against the schema when it is written into the database.

Hive, on the other hand, doesn't verify the data when it is loaded, but rather when a query is issued. This is called *schema on read*.

There are trade-offs between the two approaches. Schema on read makes for a very fast initial load, since the data does not have to be read, parsed, and serialized to disk in the database's internal format. The load operation is just a file copy or move. It is more flexible, too: consider having two schemas for the same underlying data, depending on the analysis being performed. (This is possible in Hive using external tables; see "Managed Tables and External Tables" on page 490.)

Schema on write makes query time performance faster because the database can index columns and perform compression on the data. The trade-off, however, is that it takes longer to load data into the database. Furthermore, there are many scenarios where the schema is not known at load time, so there are no indexes to apply, because the queries have not been formulated yet. These scenarios are where Hive shines.

Updates, Transactions, and Indexes

Updates, transactions, and indexes are mainstays of traditional databases. Yet, until recently, these features have not been considered a part of Hive's feature set. This is because Hive was built to operate over HDFS data using MapReduce, where full-table scans are the norm and a table update is achieved by transforming the data into a new table. For a data warehousing application that runs over large portions of the dataset, this works well.

Hive has long supported adding new rows in bulk to an existing table by using `INSERT INTO` to add new data files to a table. From release 0.14.0, finer-grained changes are possible, so you can call `INSERT INTO TABLE...VALUES` to insert small batches of values computed in SQL. In addition, it is possible to `UPDATE` and `DELETE` rows in a table.

HDFS does not provide in-place file updates, so changes resulting from inserts, updates, and deletes are stored in small delta files. Delta files are periodically merged into the base table files by MapReduce jobs that are run in the background by the metastore. These features only work in the context of transactions (introduced in Hive 0.13.0), so the table they are being used on needs to have transactions enabled on it. Queries reading the table are guaranteed to see a consistent snapshot of the table.

Hive also has support for table- and partition-level locking. Locks prevent, for example, one process from dropping a table while another is reading from it. Locks are managed transparently using ZooKeeper, so the user doesn't have to acquire or release them, although it is possible to get information about which locks are being held via the `SHOW LOCKS` statement. By default, locks are not enabled.

Hive indexes can speed up queries in certain cases. A query such as `SELECT * from t WHERE x = a`, for example, can take advantage of an index on column `x`, since only a small portion of the table's files need to be scanned. There are currently two index types: *compact* and *bitmap*. (The index implementation was designed to be pluggable, so it's expected that a variety of implementations will emerge for different use cases.)

Compact indexes store the HDFS block numbers of each value, rather than each file offset, so they don't take up much disk space but are still effective for the case where values are clustered together in nearby rows. Bitmap indexes use compressed bitsets to efficiently store the rows that a particular value appears in, and they are usually appropriate for low-cardinality columns (such as gender or country).

SQL-on-Hadoop Alternatives

In the years since Hive was created, many other SQL-on-Hadoop engines have emerged to address some of Hive's limitations. Cloudera Impala, an open source interactive SQL engine, was one of the first, giving an order of magnitude performance boost compared to Hive running on MapReduce. Impala uses a dedicated daemon that runs on each datanode in the cluster. When a client runs a query it contacts an arbitrary node running an Impala daemon, which acts as a coordinator node for the query. The coordinator sends work to other Impala daemons in the cluster and combines their results into the full result set for the query. Impala uses the Hive metastore and supports Hive formats and most HiveQL constructs (plus SQL-92), so in practice it is straightforward to migrate between the two systems, or to run both on the same cluster.

Hive has not stood still, though, and since Impala was launched, the "Stinger" initiative by Hortonworks has improved the performance of Hive through support for Tez as an execution engine, and the addition of a vectorized query engine among other improvements.

Other prominent open source Hive alternatives include Presto from Facebook, Apache Drill, and Spark SQL. Presto and Drill have similar architectures to Impala, although Drill targets SQL:2011 rather than HiveQL. Spark SQL uses Spark as its underlying engine, and lets you embed SQL queries in Spark programs.

Spark SQL is different to using the Spark execution engine from within Hive ("Hive on Spark," see "Execution engines" on page 477). Hive, on Spark provides all the features of Hive since it is a part of the Hive project. Spark SQL, on the other hand, is a new SQL engine that offers some level of Hive compatibility.

Apache Phoenix takes a different approach entirely: it provides SQL on HBase. SQL access is through a JDBC driver that turns queries into HBase scans and takes advantage of HBase coprocessors to perform server-side aggregation. Metadata is stored in HBase, too.