

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).

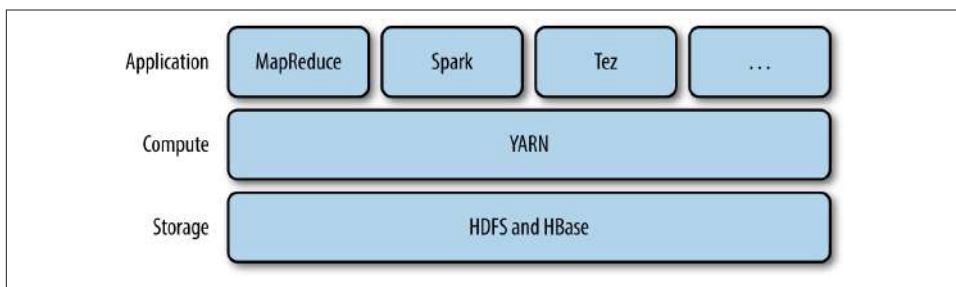


Figure 4-1. YARN applications

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.

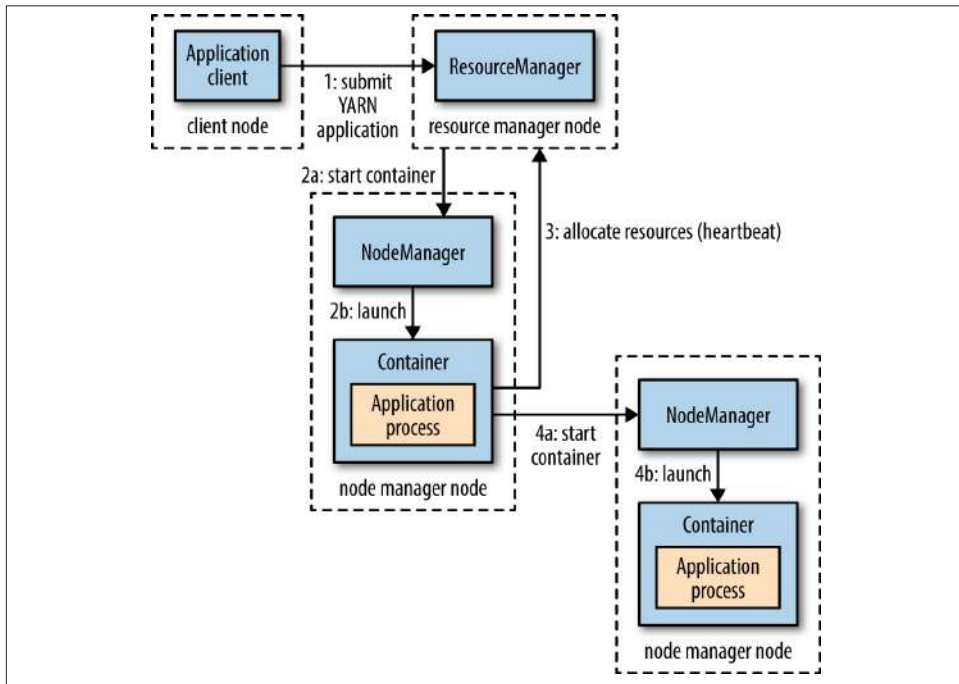


Figure 4-2. How YARN runs an application

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.

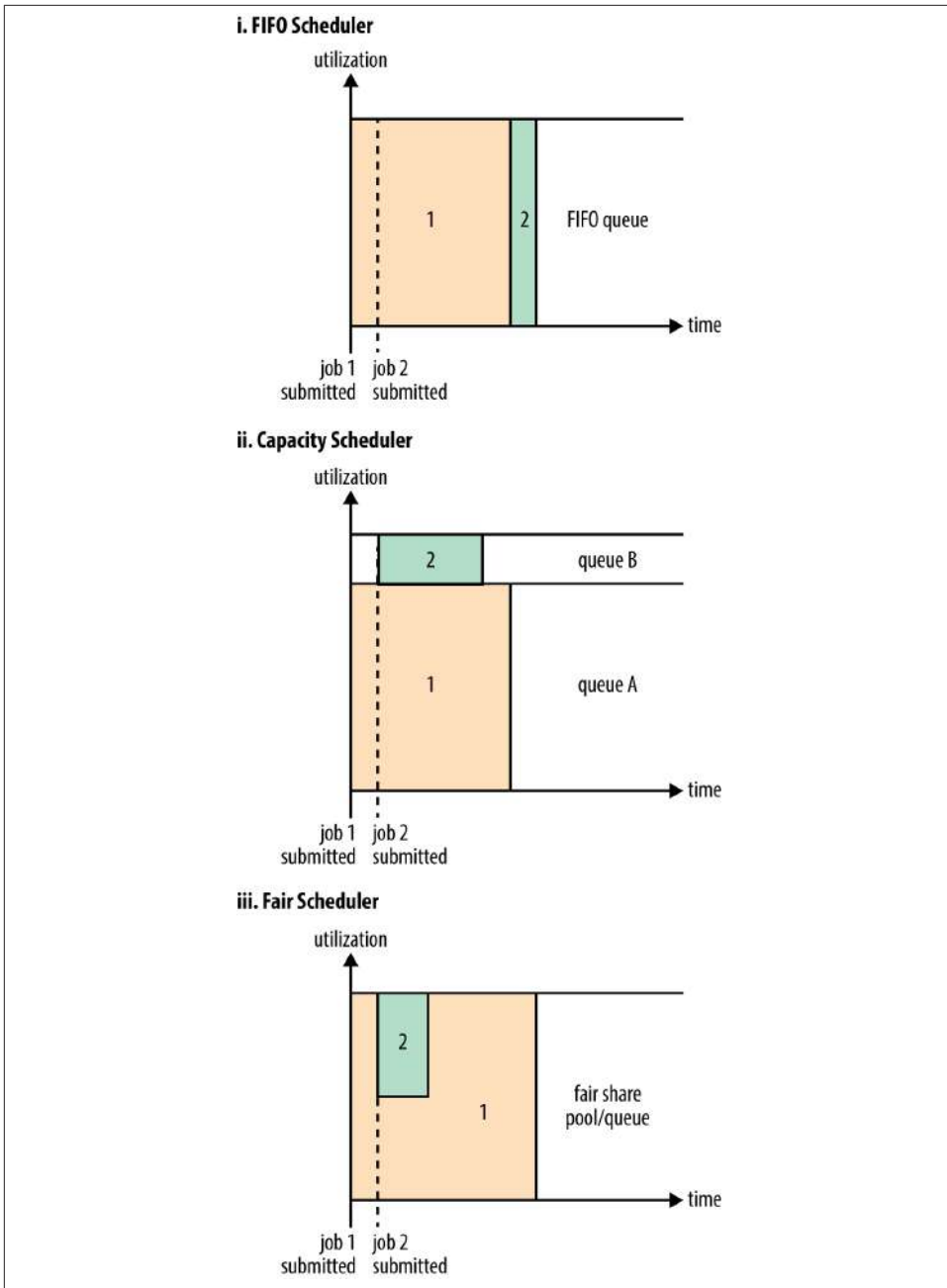


Figure 4-3. Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii)

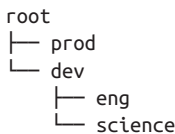
Capacity Scheduler Configuration

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:



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.queue.name` 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.

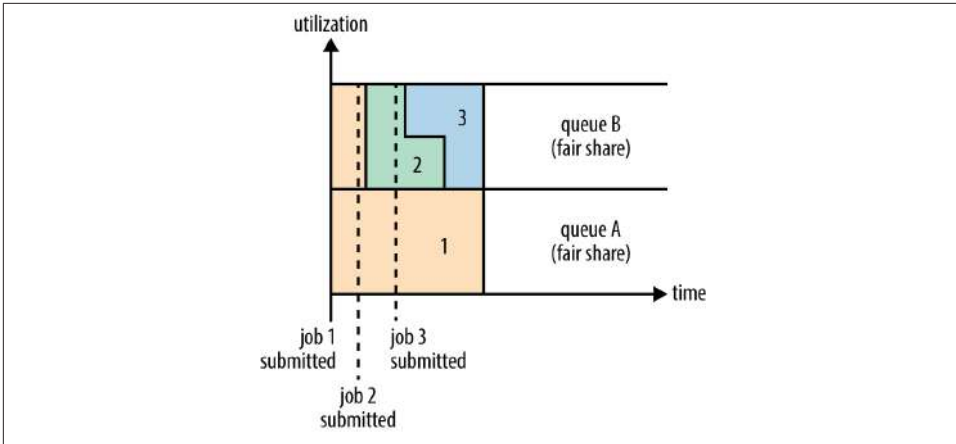


Figure 4-4. Fair sharing between user queues

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 `minSharePreemptionTimeout` 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 `fairSharePreemptionTimeout` on a queue. The threshold may also be changed from its default of 0.5 by setting `defaultFairSharePreemptionThreshold` and `fairSharePreemptionThreshold` (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).

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 (nonchecksummed) 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	<i>.deflate</i>	No
gzip	<i>gzip</i>	DEFLATE	<i>.gz</i>	No
bzip2	<i>bzip2</i>	bzip2	<i>.bz2</i>	Yes
LZO	<i>lzop</i>	LZO	<i>.lzo</i>	No ^b
LZ4	N/A	LZ4	<i>.lz4</i>	No
Snappy	N/A	Snappy	<i>.snappy</i>	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 Map-Reduce; 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 <code>CompressionCodec</code>
DEFLATE	<code>org.apache.hadoop.io.compress.DefaultCodec</code>
gzip	<code>org.apache.hadoop.io.compress.GzipCodec</code>
bzip2	<code>org.apache.hadoop.io.compress.BZip2Codec</code>
LZO	<code>com.hadoop.compression.lzo.LzopCodec</code>
LZ4	<code>org.apache.hadoop.io.compress.Lz4Codec</code>
Snappy	<code>org.apache.hadoop.io.compress.SnappyCodec</code>

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 *lzo* 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
<code>io.compression.codecs</code>	Comma-separated Class names		A list of additional <code>CompressionCodec</code> 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 π), 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
<code>mapreduce.output.fileoutputformat.compress</code>	boolean	false	Whether to compress outputs
<code>mapreduce.output.fileoutputformat.compress.codec</code>	Class name	<code>org.apache.hadoop.io.compress.DefaultCodec</code>	The compression codec to use for outputs
<code>mapreduce.output.fileoutputformat.compress.type</code>	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` put 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;
```



```

    void readFields(DataInput in) throws IOException;
}

```

Let's look at a particular Writable to see what we can do with it. We will use IntWritable, a wrapper for a Java int. We can create one and set its value using the set() method:

```

IntWritable writable = new IntWritable();
writable.set(163);

```

Equivalently, we can use the constructor that takes the integer value:

```

IntWritable writable = new IntWritable(163);

```

To examine the serialized form of the IntWritable, we write a small helper method that wraps a java.io.ByteArrayOutputStream in a java.io.DataOutputStream (an implementation of java.io.DataOutput) to capture the bytes in the serialized stream:

```

public static byte[] serialize(Writable writable) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(out);
    writable.write(dataOut);
    dataOut.close();
    return out.toByteArray();
}

```

An integer is written using four bytes (as we see using JUnit 4 assertions):

```

byte[] bytes = serialize(writable);
assertThat(bytes.length, is(4));

```

The bytes are written in big-endian order (so the most significant byte is written to the stream first, which is dictated by the java.io.DataOutput interface), and we can see their hexadecimal representation by using a method on Hadoop's StringUtils:

```

assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));

```

Let's try deserialization. Again, we create a helper method to read a Writable object from a byte array:

```

public static byte[] deserialize(Writable writable, byte[] bytes)
    throws IOException {
    ByteArrayInputStream in = new ByteArrayInputStream(bytes);
    DataInputStream dataIn = new DataInputStream(in);
    writable.readFields(dataIn);
    dataIn.close();
    return bytes;
}

```

We construct a new, value-less IntWritable, and then call deserialize() to read from the output data that we just wrote. Then we check that its value, retrieved using the get() method, is the original value, 163:

```
IntWritable newWritable = new IntWritable();
deserialize(newWritable, bytes);
assertThat(newWritable.get(), is(163));
```

WritableComparable and comparators

IntWritable implements the WritableComparable interface, which is just a subinterface of the Writable and java.lang.Comparable interfaces:

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

public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another. One optimization that Hadoop provides is the RawComparator extension of Java's Comparator:

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

import java.util.Comparator;

public interface RawComparator<T> extends Comparator<T> {

    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

}
```

This interface permits implementors to compare records read from a stream without deserializing them into objects, thereby avoiding any overhead of object creation. For example, the comparator for IntWritable implements the raw compare() method by reading an integer from each of the byte arrays b1 and b2 and comparing them directly from the given start positions (s1 and s2) and lengths (l1 and l2).

WritableComparator is a general-purpose implementation of RawComparator for WritableComparable classes. It provides two main functions. First, it provides a default implementation of the raw compare() method that deserializes the objects to be compared from the stream and invokes the object compare() method. Second, it acts as a factory for RawComparator instances (that Writable implementations have registered). For example, to obtain a comparator for IntWritable, we just use:

```
RawComparator<IntWritable> comparator =
    WritableComparator.get(IntWritable.class);
```

The comparator can be used to compare two IntWritable objects:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

or their serialized representations:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
    greaterThan(0));
```

Writable Classes

Hadoop comes with a large selection of Writable classes, which are available in the `org.apache.hadoop.io` package. They form the class hierarchy shown in [Figure 5-1](#).

Writable wrappers for Java primitives

There are Writable wrappers for all the Java primitive types (see [Table 5-7](#)) except `char` (which can be stored in an `IntWritable`). All have a `get()` and `set()` method for retrieving and storing the wrapped value.

Table 5-7. Writable wrapper classes for Java primitives

Java primitive	Writable implementation	Serialized size (bytes)
<code>boolean</code>	<code>BooleanWritable</code>	1
<code>byte</code>	<code>ByteWritable</code>	1
<code>short</code>	<code>ShortWritable</code>	2
<code>int</code>	<code>IntWritable</code>	4
	<code>VIntWritable</code>	1–5
<code>float</code>	<code>FloatWritable</code>	4
<code>long</code>	<code>LongWritable</code>	8
	<code>VLongWritable</code>	1–9
<code>double</code>	<code>DoubleWritable</code>	8

When it comes to encoding integers, there is a choice between the fixed-length formats (`IntWritable` and `LongWritable`) and the variable-length formats (`VIntWritable` and `VLongWritable`). The variable-length formats use only a single byte to encode the value if it is small enough (between `-112` and `127`, inclusive); otherwise, they use the first byte to indicate whether the value is positive or negative, and how many bytes follow. For example, `163` requires two bytes:

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.toHexString(data), is("8fa3"));
```

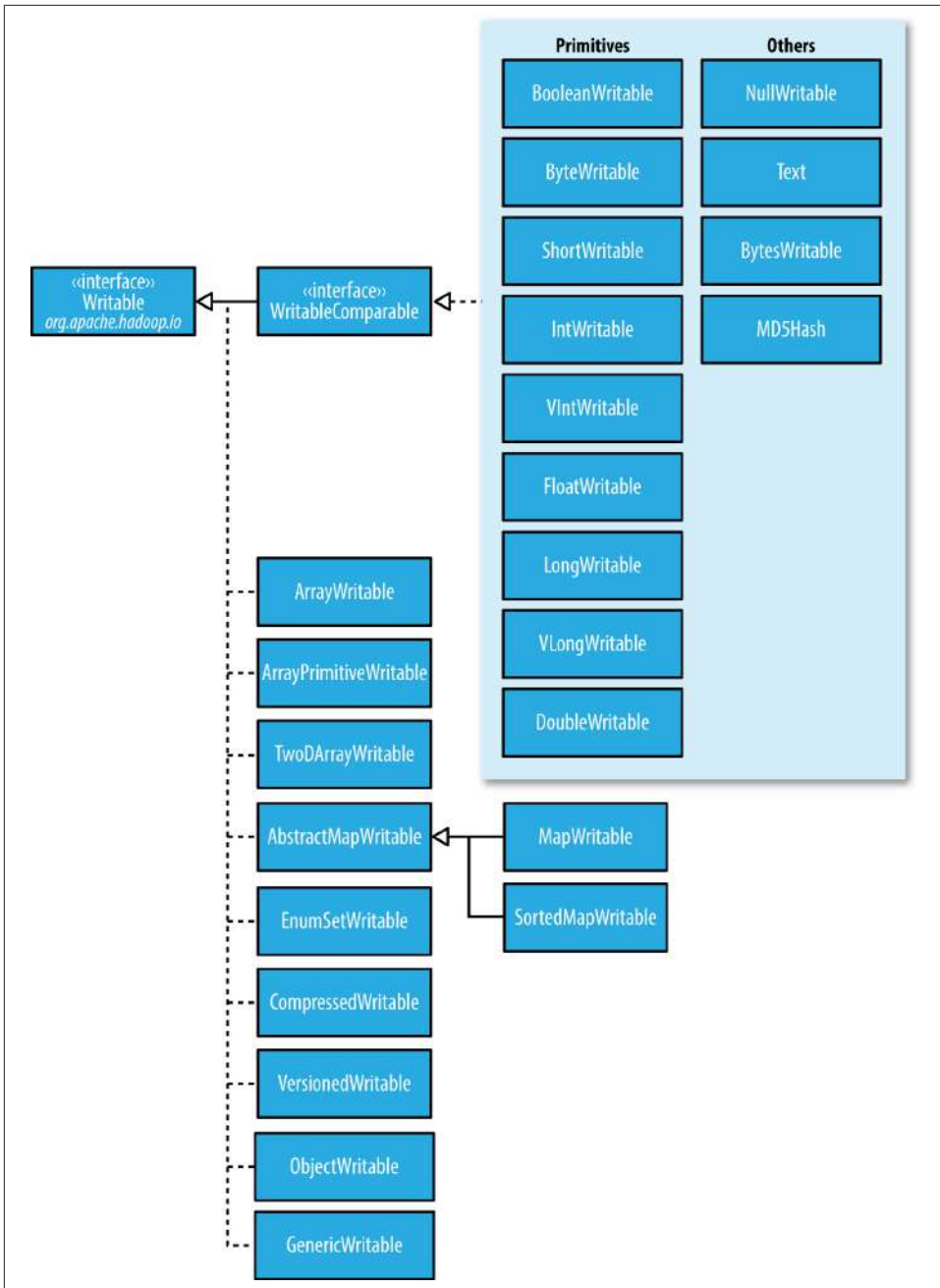


Figure 5-1. Writable class hierarchy

How do you choose between a fixed-length and a variable-length encoding? Fixed-length encodings are good when the distribution of values is fairly uniform across the whole value space, such as when using a (well-designed) hash function. Most numeric variables tend to have nonuniform distributions, though, and on average, the variable-length encoding will save space. Another advantage of variable-length encodings is that you can switch from `VIntWritable` to `VLongWritable`, because their encodings are actually the same. So, by choosing a variable-length representation, you have room to grow without committing to an 8-byte long representation from the beginning.

Text

`Text` is a `Writable` for UTF-8 sequences. It can be thought of as the `Writable` equivalent of `java.lang.String`.

The `Text` class uses an `int` (with a variable-length encoding) to store the number of bytes in the string encoding, so the maximum value is 2 GB. Furthermore, `Text` uses standard UTF-8, which makes it potentially easier to interoperate with other tools that understand UTF-8.

Indexing. Because of its emphasis on using standard UTF-8, there are some differences between `Text` and the Java `String` class. Indexing for the `Text` class is in terms of position in the encoded byte sequence, not the Unicode character in the string or the Java `char` code unit (as it is for `String`). For ASCII strings, these three concepts of index position coincide. Here is an example to demonstrate the use of the `charAt()` method:

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));

assertThat(t.charAt(2), is((int) 'd'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

Notice that `charAt()` returns an `int` representing a Unicode code point, unlike the `String` variant that returns a `char`. `Text` also has a `find()` method, which is analogous to `String`'s `indexOf()`:

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

Unicode. When we start using characters that are encoded with more than a single byte, the differences between `Text` and `String` become clear. Consider the Unicode characters shown in [Table 5-8](#).²

Table 5-8. Unicode characters

Unicode code point	U+0041	U+00DF	U+6771	U+10400
Name	LATIN CAPITAL LETTER A	LATIN SMALL LETTER SHARP S	N/A (a unified Han ideograph)	DESERET CAPITAL LETTER LONG I
UTF-8 code units	41	c3 9f	e6 9d b1	f0 90 90 80
Java representation	\u0041	\u00DF	\u6771	\uD801\uDC00

All but the last character in the table, U+10400, can be expressed using a single Java char. U+10400 is a supplementary character and is represented by two Java chars, known as a *surrogate pair*. The tests in [Example 5-5](#) show the differences between `String` and `Text` when processing a string of the four characters from [Table 5-8](#).

Example 5-5. Tests showing the differences between the `String` and `Text` classes

```
public class StringTextComparisonTest {

    @Test
    public void string() throws UnsupportedOperationException {

        String s = "\u0041\u00DF\u6771\uD801\uDC00";
        assertEquals(s.length(), is(5));
        assertEquals(s.getBytes("UTF-8").length, is(10));

        assertEquals(s.indexOf("\u0041"), is(0));
        assertEquals(s.indexOf("\u00DF"), is(1));
        assertEquals(s.indexOf("\u6771"), is(2));
        assertEquals(s.indexOf("\uD801\uDC00"), is(3));

        assertEquals(s.charAt(0), is('\u0041'));
        assertEquals(s.charAt(1), is('\u00DF'));
        assertEquals(s.charAt(2), is('\u6771'));
        assertEquals(s.charAt(3), is('\uD801'));
        assertEquals(s.charAt(4), is('\uDC00'));

        assertEquals(s.codePointAt(0), is(0x0041));
        assertEquals(s.codePointAt(1), is(0x00DF));
        assertEquals(s.codePointAt(2), is(0x6771));
        assertEquals(s.codePointAt(3), is(0x10400));
    }
}
```

2. This example is based on one from Norbert Lindenberg and Masayoshi Okutsu's "Supplementary Characters in the Java Platform," May 2004.

```

@Test
public void text() {

    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
    assertEquals(t.getLength(), 10);

    assertEquals(t.find("\u0041"), 0);
    assertEquals(t.find("\u00DF"), 1);
    assertEquals(t.find("\u6771"), 3);
    assertEquals(t.find("\uD801\uDC00"), 6);

    assertEquals(t.charAt(0), 0x0041);
    assertEquals(t.charAt(1), 0x00DF);
    assertEquals(t.charAt(3), 0x6771);
    assertEquals(t.charAt(6), 0x10400);
}
}

```

The test confirms that the length of a `String` is the number of char code units it contains (five, made up of one from each of the first three characters in the string and a surrogate pair from the last), whereas the length of a `Text` object is the number of bytes in its UTF-8 encoding (10 = 1+2+3+4). Similarly, the `indexOf()` method in `String` returns an index in char code units, and `find()` for `Text` returns a byte offset.

The `charAt()` method in `String` returns the char code unit for the given index, which in the case of a surrogate pair will not represent a whole Unicode character. The `codePointAt()` method, indexed by char code unit, is needed to retrieve a single Unicode character represented as an `int`. In fact, the `charAt()` method in `Text` is more like the `codePointAt()` method than its namesake in `String`. The only difference is that it is indexed by byte offset.

Iteration. Iterating over the Unicode characters in `Text` is complicated by the use of byte offsets for indexing, since you can't just increment the index. The idiom for iteration is a little obscure (see [Example 5-6](#)): turn the `Text` object into a `java.nio.ByteBuffer`, then repeatedly call the `bytesToCodePoint()` static method on `Text` with the buffer. This method extracts the next code point as an `int` and updates the position in the buffer. The end of the string is detected when `bytesToCodePoint()` returns `-1`.

Example 5-6. Iterating over the characters in a `Text` object

```

public class TextIterator {

    public static void main(String[] args) {
        Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");

        ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
        int cp;
        while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) {

```

```

        System.out.println(Integer.toHexString(cp));
    }
}
}

```

Running the program prints the code points for the four characters in the string:

```

% hadoop TextIterator
41
df
6771
10400

```

Mutability. Another difference from `String` is that `Text` is mutable (like all `Writable` implementations in Hadoop, except `NullWritable`, which is a singleton). You can reuse a `Text` instance by calling one of the `set()` methods on it. For example:

```

Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));

```



In some situations, the byte array returned by the `getBytes()` method may be longer than the length returned by `getLength()`:

```

Text t = new Text("hadoop");
t.set(new Text("pig"));
assertThat(t.getLength(), is(3));
assertThat("Byte length not shortened", t.getBytes().length,
    is(6));

```

This shows why it is imperative that you always call `getLength()` when calling `getBytes()`, so you know how much of the byte array is valid data.

Resorting to `String`. `Text` doesn't have as rich an API for manipulating strings as `java.lang.String`, so in many cases, you need to convert the `Text` object to a `String`. This is done in the usual way, using the `toString()` method:

```

assertThat(new Text("hadoop").toString(), is("hadoop"));

```

BytesWritable

`BytesWritable` is a wrapper for an array of binary data. Its serialized format is a 4-byte integer field that specifies the number of bytes to follow, followed by the bytes themselves. For example, the byte array of length 2 with values 3 and 5 is serialized as a 4-byte integer (00000002) followed by the two bytes from the array (03 and 05):


```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

BytesWritable is mutable, and its value may be changed by calling its `set()` method. As with Text, the size of the byte array returned from the `getBytes()` method for BytesWritable—the capacity—may not reflect the actual size of the data stored in the BytesWritable. You can determine the size of the BytesWritable by calling `getLength()`. To demonstrate:

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

NullWritable

NullWritable is a special type of Writable, as it has a zero-length serialization. No bytes are written to or read from the stream. It is used as a placeholder; for example, in Map-Reduce, a key or a value can be declared as a NullWritable when you don't need to use that position, effectively storing a constant empty value. NullWritable can also be useful as a key in a SequenceFile when you want to store a list of values, as opposed to key-value pairs. It is an immutable singleton, and the instance can be retrieved by calling `NullWritable.get()`.

ObjectWritable and GenericWritable

ObjectWritable is a general-purpose wrapper for the following: Java primitives, String, enum, Writable, null, or arrays of any of these types. It is used in Hadoop RPC to marshal and unmarshal method arguments and return types.

ObjectWritable is useful when a field can be of more than one type. For example, if the values in a SequenceFile have multiple types, you can declare the value type as an ObjectWritable and wrap each type in an ObjectWritable. Being a general-purpose mechanism, it wastes a fair amount of space because it writes the classname of the wrapped type every time it is serialized. In cases where the number of types is small and known ahead of time, this can be improved by having a static array of types and using the index into the array as the serialized reference to the type. This is the approach that GenericWritable takes, and you have to subclass it to specify which types to support.

Writable collections

The `org.apache.hadoop.io` package includes six Writable collection types: ArrayWritable, ArrayPrimitiveWritable, TwoArrayWritable, MapWritable, SortedMapWritable, and EnumSetWritable.

ArrayWritable and TwoArrayWritable are Writable implementations for arrays and two-dimensional arrays (array of arrays) of Writable instances. All the elements of an

ArrayWritable or a TwoArrayWritable must be instances of the same class, which is specified at construction as follows:

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

In contexts where the Writable is defined by type, such as in SequenceFile keys or values or as input to MapReduce in general, you need to subclass ArrayWritable (or TwoArrayWritable, as appropriate) to set the type statically. For example:

```
public class TextArrayWritable extends ArrayWritable {
    public TextArrayWritable() {
        super(Text.class);
    }
}
```

ArrayWritable and TwoArrayWritable both have get() and set() methods, as well as a toArray() method, which creates a shallow copy of the array (or 2D array).

ArrayPrimitiveWritable is a wrapper for arrays of Java primitives. The component type is detected when you call set(), so there is no need to subclass to set the type.

MapWritable is an implementation of java.util.Map<Writable, Writable>, and SortedMapWritable is an implementation of java.util.SortedMap<WritableComparable, Writable>. The type of each key and value field is a part of the serialization format for that field. The type is stored as a single byte that acts as an index into an array of types. The array is populated with the standard types in the org.apache.hadoop.io package, but custom Writable types are accommodated, too, by writing a header that encodes the type array for nonstandard types. As they are implemented, MapWritable and SortedMapWritable use positive byte values for custom types, so a maximum of 127 distinct nonstandard Writable classes can be used in any particular MapWritable or SortedMapWritable instance. Here's a demonstration of using a MapWritable with different types for keys and values:

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));

MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)),
    is(new LongWritable(163)));
```

Conspicuous by their absence are Writable collection implementations for sets and lists. A general set can be emulated by using a MapWritable (or a SortedMapWritable for a sorted set) with NullWritable values. There is also EnumSetWritable for sets of enum types. For lists of a single type of Writable, ArrayWritable is adequate, but to store different types of Writable in a single list, you can use GenericWritable to wrap

the elements in an `ArrayWritable`. Alternatively, you could write a general `ListWritable` using the ideas from `MapWritable`.

Implementing a Custom Writable

Hadoop comes with a useful set of `Writable` implementations that serve most purposes; however, on occasion, you may need to write your own custom implementation. With a custom `Writable`, you have full control over the binary representation and the sort order. Because `Writables` are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance. The stock `Writable` implementations that come with Hadoop are well tuned, but for more elaborate structures, it is often better to create a new `Writable` type rather than composing the stock types.



If you are considering writing a custom `Writable`, it may be worth trying another serialization framework, like Avro, that allows you to define custom types declaratively. See “[Serialization Frameworks](#)” on [page 126](#) and [Chapter 12](#).

To demonstrate how to create a custom `Writable`, we shall write an implementation that represents a pair of strings, called `TextPair`. The basic implementation is shown in [Example 5-7](#).

Example 5-7. A `Writable` implementation that stores a pair of `Text` objects

```
import java.io.*;

import org.apache.hadoop.io.*;

public class TextPair implements WritableComparable<TextPair> {

    private Text first;
    private Text second;

    public TextPair() {
        set(new Text(), new Text());
    }

    public TextPair(String first, String second) {
        set(new Text(first), new Text(second));
    }

    public TextPair(Text first, Text second) {
        set(first, second);
    }

    public void set(Text first, Text second) {
```

```

        this.first = first;
        this.second = second;
    }

    public Text getFirst() {
        return first;
    }

    public Text getSecond() {
        return second;
    }

    @Override
    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }

    @Override
    public void readFields(DataInput in) throws IOException {
        first.readFields(in);
        second.readFields(in);
    }

    @Override
    public int hashCode() {
        return first.hashCode() * 163 + second.hashCode();
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof TextPair) {
            TextPair tp = (TextPair) o;
            return first.equals(tp.first) && second.equals(tp.second);
        }
        return false;
    }

    @Override
    public String toString() {
        return first + "\t" + second;
    }

    @Override
    public int compareTo(TextPair tp) {
        int cmp = first.compareTo(tp.first);
        if (cmp != 0) {
            return cmp;
        }
        return second.compareTo(tp.second);
    }
}

```

The first part of the implementation is straightforward: there are two `Text` instance variables, `first` and `second`, and associated constructors, getters, and setters. All `Writable` implementations must have a default constructor so that the MapReduce framework can instantiate them, then populate their fields by calling `readFields()`. `Writable` instances are mutable and often reused, so you should take care to avoid allocating objects in the `write()` or `readFields()` methods.

`TextPair`'s `write()` method serializes each `Text` object in turn to the output stream by delegating to the `Text` objects themselves. Similarly, `readFields()` deserializes the bytes from the input stream by delegating to each `Text` object. The `DataOutput` and `DataInput` interfaces have a rich set of methods for serializing and deserializing Java primitives, so, in general, you have complete control over the wire format of your `Writable` object.

Just as you would for any value object you write in Java, you should override the `hashCode()`, `equals()`, and `toString()` methods from `java.lang.Object`. The `hashCode()` method is used by the `HashPartitioner` (the default partitioner in MapReduce) to choose a reduce partition, so you should make sure that you write a good hash function that mixes well to ensure reduce partitions are of a similar size.



If you plan to use your custom `Writable` with `TextOutputFormat`, you must implement its `toString()` method. `TextOutputFormat` calls `toString()` on keys and values for their output representation. For `TextPair`, we write the underlying `Text` objects as strings separated by a tab character.

`TextPair` is an implementation of `WritableComparable`, so it provides an implementation of the `compareTo()` method that imposes the ordering you would expect: it sorts by the first string followed by the second. Notice that, apart from the number of `Text` objects it can store, `TextPair` differs from `TextArrayWritable` (which we discussed in the previous section), since `TextArrayWritable` is only a `Writable`, not a `WritableComparable`.

Implementing a `RawComparator` for speed

The code for `TextPair` in [Example 5-7](#) will work as it stands; however, there is a further optimization we can make. As explained in [“WritableComparable and comparators” on page 112](#), when `TextPair` is being used as a key in MapReduce, it will have to be deserialized into an object for the `compareTo()` method to be invoked. What if it were possible to compare two `TextPair` objects just by looking at their serialized representations?

It turns out that we can do this because `TextPair` is the concatenation of two `Text` objects, and the binary representation of a `Text` object is a variable-length integer containing the number of bytes in the UTF-8 representation of the string, followed by the

UTF-8 bytes themselves. The trick is to read the initial length so we know how long the first Text object's byte representation is; then we can delegate to Text's RawComparator and invoke it with the appropriate offsets for the first or second string. [Example 5-8](#) gives the details (note that this code is nested in the TextPair class).

Example 5-8. A RawComparator for comparing TextPair byte representations

```
public static class Comparator extends WritableComparator {

    private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();

    public Comparator() {
        super(TextPair.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1,
                      byte[] b2, int s2, int l2) {

        try {
            int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
            int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
            int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
            if (cmp != 0) {
                return cmp;
            }
            return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,
                                           b2, s2 + firstL2, l2 - firstL2);
        } catch (IOException e) {
            throw new IllegalArgumentException(e);
        }
    }
}

static {
    WritableComparator.define(TextPair.class, new Comparator());
}
```

We actually subclass WritableComparator rather than implementing RawComparator directly, since it provides some convenience methods and default implementations. The subtle part of this code is calculating firstL1 and firstL2, the lengths of the first Text field in each byte stream. Each is made up of the length of the variable-length integer (returned by decodeVIntSize() on WritableUtils) and the value it is encoding (returned by readVInt()).

The static block registers the raw comparator so that whenever MapReduce sees the TextPair class, it knows to use the raw comparator as its default comparator.

Custom comparators

As you can see with `TextPair`, writing raw comparators takes some care because you have to deal with details at the byte level. It is worth looking at some of the implementations of `Writable` in the `org.apache.hadoop.io` package for further ideas if you need to write your own. The utility methods on `WritableUtils` are very handy, too.

Custom comparators should also be written to be `RawComparators`, if possible. These are comparators that implement a different sort order from the natural sort order defined by the default comparator. [Example 5-9](#) shows a comparator for `TextPair`, called `FirstComparator`, that considers only the first string of the pair. Note that we override the `compare()` method that takes objects so both `compare()` methods have the same semantics.

We will make use of this comparator in [Chapter 9](#), when we look at joins and secondary sorting in MapReduce (see “[Joins](#)” on page 268).

Example 5-9. A custom `RawComparator` for comparing the first field of `TextPair` byte representations

```
public static class FirstComparator extends WritableComparator {

    private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator();

    public FirstComparator() {
        super(TextPair.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1,
                      byte[] b2, int s2, int l2) {

        try {
            int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
            int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
            return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
        } catch (IOException e) {
            throw new IllegalArgumentException(e);
        }
    }

    @Override
    public int compare(WritableComparable a, WritableComparable b) {
        if (a instanceof TextPair && b instanceof TextPair) {
            return ((TextPair) a).first.compareTo(((TextPair) b).first);
        }
        return super.compare(a, b);
    }
}
```

Serialization Frameworks

Although most MapReduce programs use `Writable` key and value types, this isn't mandated by the MapReduce API. In fact, any type can be used; the only requirement is a mechanism that translates to and from a binary representation of each type.

To support this, Hadoop has an API for pluggable serialization frameworks. A serialization framework is represented by an implementation of `Serialization` (in the `org.apache.hadoop.io.serializer` package). `WritableSerialization`, for example, is the implementation of `Serialization` for `Writable` types.

A `Serialization` defines a mapping from types to `Serializer` instances (for turning an object into a byte stream) and `Deserializer` instances (for turning a byte stream into an object).

Set the `io.serialization` property to a comma-separated list of classnames in order to register `Serialization` implementations. Its default value includes `org.apache.hadoop.io.serializer.WritableSerialization` and the Avro Specific and Reflect serializations (see “[Avro Data Types and Schemas](#)” on page 346), which means that only `Writable` or Avro objects can be serialized or deserialized out of the box.

Hadoop includes a class called `JavaSerialization` that uses Java Object Serialization. Although it makes it convenient to be able to use standard Java types such as `Integer` or `String` in MapReduce programs, Java Object Serialization is not as efficient as `Writables`, so it's not worth making this trade-off (see the following sidebar).

Why Not Use Java Object Serialization?

Java comes with its own serialization mechanism, called Java Object Serialization (often referred to simply as “Java Serialization”), that is tightly integrated with the language, so it's natural to ask why this wasn't used in Hadoop. Here's what Doug Cutting said in response to that question:

Why didn't I use `Serialization` when we first started Hadoop? Because it looked big and hairy and I thought we needed something lean and mean, where we had precise control over exactly how objects are written and read, since that is central to Hadoop. With `Serialization` you can get some control, but you have to fight for it.

The logic for not using RMI [Remote Method Invocation] was similar. Effective, high-performance inter-process communications are critical to Hadoop. I felt like we'd need to precisely control how things like connections, timeouts and buffers are handled, and RMI gives you little control over those.

The problem is that Java Serialization doesn't meet the criteria for a serialization format listed earlier: compact, fast, extensible, and interoperable.

Serialization IDL

There are a number of other serialization frameworks that approach the problem in a different way: rather than defining types through code, you define them in a language-neutral, declarative fashion, using an *interface description language* (IDL). The system can then generate types for different languages, which is good for interoperability. They also typically define versioning schemes that make type evolution straightforward.

[Apache Thrift](#) and [Google Protocol Buffers](#) are both popular serialization frameworks, and both are commonly used as a format for persistent binary data. There is limited support for these as MapReduce formats;³ however, they are used internally in parts of Hadoop for RPC and data exchange.

Avro is an IDL-based serialization framework designed to work well with large-scale data processing in Hadoop. It is covered in [Chapter 12](#).

File-Based Data Structures

For some applications, you need a specialized data structure to hold your data. For doing MapReduce-based processing, putting each blob of binary data into its own file doesn't scale, so Hadoop developed a number of higher-level containers for these situations.

SequenceFile

Imagine a logfile where each log record is a new line of text. If you want to log binary types, plain text isn't a suitable format. Hadoop's `SequenceFile` class fits the bill in this situation, providing a persistent data structure for binary key-value pairs. To use it as a logfile format, you would choose a key, such as timestamp represented by a `LongWritable`, and the value would be a `Writable` that represents the quantity being logged.

`SequenceFiles` also work well as containers for smaller files. HDFS and MapReduce are optimized for large files, so packing files into a `SequenceFile` makes storing and processing the smaller files more efficient ("[Processing a whole file as a record](#)" on [page 228](#) contains a program to pack files into a `SequenceFile`).⁴

Writing a SequenceFile

To create a `SequenceFile`, use one of its `createWriter()` static methods, which return a `SequenceFile.Writer` instance. There are several overloaded versions, but they all require you to specify a stream to write to (either an `FSDataOutputStream` or a

3. Twitter's [Elephant Bird project](#) includes tools for working with Thrift and Protocol Buffers in Hadoop.

4. In a similar vein, the blog post "[A Million Little Files](#)" by Stuart Sierra includes code for converting a tar file into a `SequenceFile`.

FileSystem and Path pairing), a Configuration object, and the key and value types. Optional arguments include the compression type and codec, a Progressable callback to be informed of write progress, and a Metadata instance to be stored in the SequenceFile header.

The keys and values stored in a SequenceFile do not necessarily need to be Writables. Any types that can be serialized and deserialized by a Serialization may be used.

Once you have a SequenceFile.Writer, you then write key-value pairs using the append() method. When you've finished, you call the close() method (SequenceFile.Writer implements java.io.Closeable).

Example 5-10 shows a short program to write some key-value pairs to a Sequence File using the API just described.

Example 5-10. Writing a SequenceFile

```
public class SequenceFileWriteDemo {

    private static final String[] DATA = {
        "One, two, buckle my shoe",
        "Three, four, shut the door",
        "Five, six, pick up sticks",
        "Seven, eight, lay them straight",
        "Nine, ten, a big fat hen"
    };

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

        IntWritable key = new IntWritable();
        Text value = new Text();
        SequenceFile.Writer writer = null;
        try {
            writer = SequenceFile.createWriter(fs, conf, path,
                key.getClass(), value.getClass());

            for (int i = 0; i < 100; i++) {
                key.set(100 - i);
                value.set(DATA[i % DATA.length]);
                System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
                writer.append(key, value);
            }
        } finally {
            IOUtils.closeStream(writer);
        }
    }
}
```

The keys in the sequence file are integers counting down from 100 to 1, represented as `IntWritable` objects. The values are `Text` objects. Before each record is appended to the `SequenceFile.Writer`, we call the `getLength()` method to discover the current position in the file. (We will use this information about record boundaries in the next section, when we read the file nonsequentially.) We write the position out to the console, along with the key and value pairs. The result of running it is shown here:

```
% hadoop SequenceFileWriteDemo numbers.seq
[128] 100    One, two, buckle my shoe
[173] 99     Three, four, shut the door
[220] 98     Five, six, pick up sticks
[264] 97     Seven, eight, lay them straight
[314] 96     Nine, ten, a big fat hen
[359] 95     One, two, buckle my shoe
[404] 94     Three, four, shut the door
[451] 93     Five, six, pick up sticks
[495] 92     Seven, eight, lay them straight
[545] 91     Nine, ten, a big fat hen
...
[1976] 60    One, two, buckle my shoe
[2021] 59    Three, four, shut the door
[2088] 58    Five, six, pick up sticks
[2132] 57    Seven, eight, lay them straight
[2182] 56    Nine, ten, a big fat hen
...
[4557] 5     One, two, buckle my shoe
[4602] 4     Three, four, shut the door
[4649] 3     Five, six, pick up sticks
[4693] 2     Seven, eight, lay them straight
[4743] 1     Nine, ten, a big fat hen
```

Reading a SequenceFile

Reading sequence files from beginning to end is a matter of creating an instance of `SequenceFile.Reader` and iterating over records by repeatedly invoking one of the `next()` methods. Which one you use depends on the serialization framework you are using. If you are using `Writable` types, you can use the `next()` method that takes a key and a value argument and reads the next key and value in the stream into these variables:

```
public boolean next(Writable key, Writable val)
```

The return value is `true` if a key-value pair was read and `false` if the end of the file has been reached.

For other, non-`Writable` serialization frameworks (such as Apache Thrift), you should use these two methods:

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

In this case, you need to make sure that the serialization you want to use has been set in the `io.serializations` property; see “[Serialization Frameworks](#)” on page 126.

If the `next()` method returns a non-null object, a key-value pair was read from the stream, and the value can be retrieved using the `getCurrentValue()` method. Otherwise, if `next()` returns null, the end of the file has been reached.

The program in [Example 5-11](#) demonstrates how to read a sequence file that has Writable keys and values. Note how the types are discovered from the Sequence File.Reader via calls to `getKeyClass()` and `getValueClass()`, and then `ReflectionUtils` is used to create an instance for the key and an instance for the value. This technique allows the program to be used with any sequence file that has Writable keys and values.

Example 5-11. Reading a SequenceFile

```
public class SequenceFileReadDemo {

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

        SequenceFile.Reader reader = null;
        try {
            reader = new SequenceFile.Reader(fs, path, conf);
            Writable key = (Writable)
                ReflectionUtils.newInstance(reader.getKeyClass(), conf);
            Writable value = (Writable)
                ReflectionUtils.newInstance(reader.getValueClass(), conf);
            long position = reader.getPosition();
            while (reader.next(key, value)) {
                String syncSeen = reader.syncSeen() ? "*" : "";
                System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
                position = reader.getPosition(); // beginning of next record
            }
        } finally {
            IOUtils.closeStream(reader);
        }
    }
}
```

Another feature of the program is that it displays the positions of the *sync points* in the sequence file. A sync point is a point in the stream that can be used to resynchronize with a record boundary if the reader is “lost”—for example, after seeking to an arbitrary position in the stream. Sync points are recorded by `SequenceFile.Writer`, which inserts a special entry to mark the sync point every few records as a sequence file is being

written. Such entries are small enough to incur only a modest storage overhead—less than 1%. Sync points always align with record boundaries.

Running the program in [Example 5-11](#) shows the sync points in the sequence file as asterisks. The first one occurs at position 2021 (the second one occurs at position 4075, but is not shown in the output):

```
% hadoop SequenceFileReadDemo numbers.seq
[128] 100 One, two, buckle my shoe
[173] 99 Three, four, shut the door
[220] 98 Five, six, pick up sticks
[264] 97 Seven, eight, lay them straight
[314] 96 Nine, ten, a big fat hen
[359] 95 One, two, buckle my shoe
[404] 94 Three, four, shut the door
[451] 93 Five, six, pick up sticks
[495] 92 Seven, eight, lay them straight
[545] 91 Nine, ten, a big fat hen
[590] 90 One, two, buckle my shoe
...
[1976] 60 One, two, buckle my shoe
[2021*] 59 Three, four, shut the door
[2088] 58 Five, six, pick up sticks
[2132] 57 Seven, eight, lay them straight
[2182] 56 Nine, ten, a big fat hen
...
[4557] 5 One, two, buckle my shoe
[4602] 4 Three, four, shut the door
[4649] 3 Five, six, pick up sticks
[4693] 2 Seven, eight, lay them straight
[4743] 1 Nine, ten, a big fat hen
```

There are two ways to seek to a given position in a sequence file. The first is the `seek()` method, which positions the reader at the given point in the file. For example, seeking to a record boundary works as expected:

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(95));
```

But if the position in the file is not at a record boundary, the reader fails when the `next()` method is called:

```
reader.seek(360);
reader.next(key, value); // fails with IOException
```

The second way to find a record boundary makes use of sync points. The `sync(long position)` method on `SequenceFile.Reader` positions the reader at the next sync point after `position`. (If there are no sync points in the file after this position, then the reader will be positioned at the end of the file.) Thus, we can call `sync()` with any position in

the stream—not necessarily a record boundary—and the reader will reestablish itself at the next sync point so reading can continue:

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```



`SequenceFile.Writer` has a method called `sync()` for inserting a sync point at the current position in the stream. This is not to be confused with the `hsync()` method defined by the `Syncable` interface for synchronizing buffers to the underlying device (see “[Coherency Model](#)” on page 74).

Sync points come into their own when using sequence files as input to MapReduce, since they permit the files to be split and different portions to be processed independently by separate map tasks (see “[SequenceFileInputFormat](#)” on page 236).

Displaying a SequenceFile with the command-line interface

The `hadoop fs` command has a `-text` option to display sequence files in textual form. It looks at a file’s magic number so that it can attempt to detect the type of the file and appropriately convert it to text. It can recognize gzipped files, sequence files, and Avro datafiles; otherwise, it assumes the input is plain text.

For sequence files, this command is really useful only if the keys and values have meaningful string representations (as defined by the `toString()` method). Also, if you have your own key or value classes, you will need to make sure they are on Hadoop’s classpath.

Running it on the sequence file we created in the previous section gives the following output:

```
% hadoop fs -text numbers.seq | head
100    One, two, buckle my shoe
99     Three, four, shut the door
98     Five, six, pick up sticks
97     Seven, eight, lay them straight
96     Nine, ten, a big fat hen
95     One, two, buckle my shoe
94     Three, four, shut the door
93     Five, six, pick up sticks
92     Seven, eight, lay them straight
91     Nine, ten, a big fat hen
```

Sorting and merging SequenceFiles

The most powerful way of sorting (and merging) one or more sequence files is to use MapReduce. MapReduce is inherently parallel and will let you specify the number of

reducers to use, which determines the number of output partitions. For example, by specifying one reducer, you get a single output file. We can use the sort example that comes with Hadoop by specifying that the input and output are sequence files and by setting the key and value types:

```
% hadoop jar \  
  $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \  
  sort -r 1 \  
    -inFormat org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat \  
    -outFormat org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat \  
    -outKey org.apache.hadoop.io.IntWritable \  
    -outValue org.apache.hadoop.io.Text \  
    numbers.seq sorted  
% hadoop fs -text sorted/part-r-00000 | head  
1      Nine, ten, a big fat hen  
2      Seven, eight, lay them straight  
3      Five, six, pick up sticks  
4      Three, four, shut the door  
5      One, two, buckle my shoe  
6      Nine, ten, a big fat hen  
7      Seven, eight, lay them straight  
8      Five, six, pick up sticks  
9      Three, four, shut the door  
10     One, two, buckle my shoe
```

Sorting is covered in more detail in [“Sorting” on page 255](#).

An alternative to using MapReduce for sort/merge is the `SequenceFile.Sorter` class, which has a number of `sort()` and `merge()` methods. These functions predate MapReduce and are lower-level functions than MapReduce (for example, to get parallelism, you need to partition your data manually), so in general MapReduce is the preferred approach to sort and merge sequence files.

The SequenceFile format

A sequence file consists of a header followed by one or more records (see [Figure 5-2](#)). The first three bytes of a sequence file are the bytes SEQ, which act as a magic number; these are followed by a single byte representing the version number. The header contains other fields, including the names of the key and value classes, compression details, user-defined metadata, and the sync marker.⁵ Recall that the sync marker is used to allow a reader to synchronize to a record boundary from any position in the file. Each file has a randomly generated sync marker, whose value is stored in the header. Sync markers appear between records in the sequence file. They are designed to incur less than a 1% storage overhead, so they don't necessarily appear between every pair of records (such is the case for short records).

5. Full details of the format of these fields may be found in `SequenceFile`'s [documentation](#) and source code.

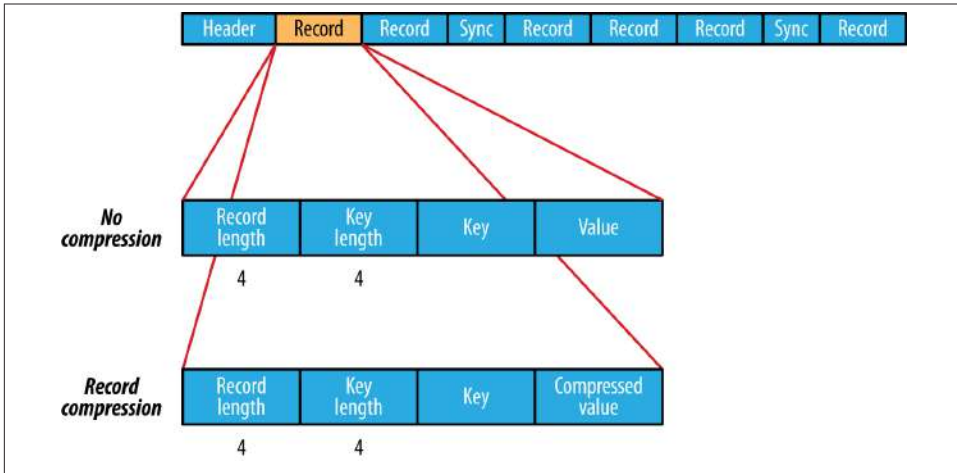


Figure 5-2. The internal structure of a sequence file with no compression and with record compression

The internal format of the records depends on whether compression is enabled, and if it is, whether it is record compression or block compression.

If no compression is enabled (the default), each record is made up of the record length (in bytes), the key length, the key, and then the value. The length fields are written as 4-byte integers adhering to the contract of the `writeInt()` method of `java.io.DataOutput`. Keys and values are serialized using the `Serialization` defined for the class being written to the sequence file.

The format for record compression is almost identical to that for no compression, except the value bytes are compressed using the codec defined in the header. Note that keys are not compressed.

Block compression (Figure 5-3) compresses multiple records at once; it is therefore more compact than and should generally be preferred over record compression because it has the opportunity to take advantage of similarities between records. Records are added to a block until it reaches a minimum size in bytes, defined by the `io.seqfile.compress.blocksize` property; the default is one million bytes. A sync marker is written before the start of every block. The format of a block is a field indicating the number of records in the block, followed by four compressed fields: the key lengths, the keys, the value lengths, and the values.

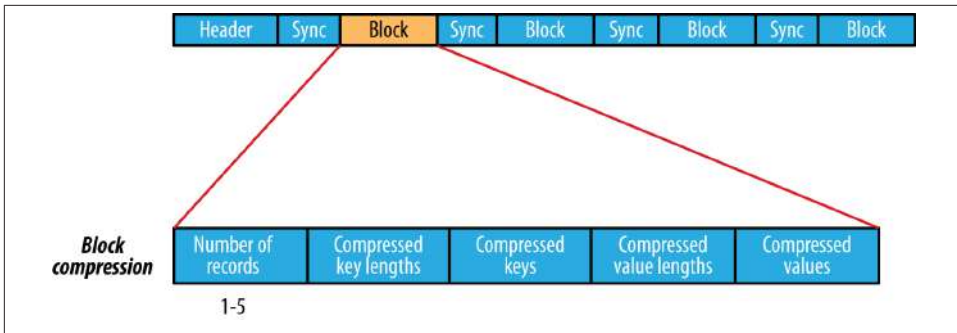


Figure 5-3. The internal structure of a sequence file with block compression

MapFile

A `MapFile` is a sorted `SequenceFile` with an index to permit lookups by key. The index is itself a `SequenceFile` that contains a fraction of the keys in the map (every 128th key, by default). The idea is that the index can be loaded into memory to provide fast lookups from the main data file, which is another `SequenceFile` containing all the map entries in sorted key order.

`MapFile` offers a very similar interface to `SequenceFile` for reading and writing—the main thing to be aware of is that when writing using `MapFile.Writer`, map entries must be added in order, otherwise an `IOException` will be thrown.

MapFile variants

Hadoop comes with a few variants on the general key-value `MapFile` interface:

- `SetFile` is a specialization of `MapFile` for storing a set of `Writable` keys. The keys must be added in sorted order.
- `ArrayFile` is a `MapFile` where the key is an integer representing the index of the element in the array and the value is a `Writable` value.
- `BloomMapFile` is a `MapFile` that offers a fast version of the `get()` method, especially for sparsely populated files. The implementation uses a dynamic Bloom filter for testing whether a given key is in the map. The test is very fast because it is in-memory, and it has a nonzero probability of false positives. Only if the test passes (the key is present) is the regular `get()` method called.

Other File Formats and Column-Oriented Formats

While sequence files and map files are the oldest binary file formats in Hadoop, they are not the only ones, and in fact there are better alternatives that should be considered for new projects.

Avro datafiles (covered in “[Avro Datafiles](#)” on page 352) are like sequence files in that they are designed for large-scale data processing—they are compact and splittable—but they are portable across different programming languages. Objects stored in Avro datafiles are described by a schema, rather than in the Java code of the implementation of a `Writable` object (as is the case for sequence files), making them very Java-centric. Avro datafiles are widely supported across components in the Hadoop ecosystem, so they are a good default choice for a binary format.

Sequence files, map files, and Avro datafiles are all row-oriented file formats, which means that the values for each row are stored contiguously in the file. In a column-oriented format, the rows in a file (or, equivalently, a table in Hive) are broken up into row splits, then each split is stored in column-oriented fashion: the values for each row in the first column are stored first, followed by the values for each row in the second column, and so on. This is shown diagrammatically in [Figure 5-4](#).

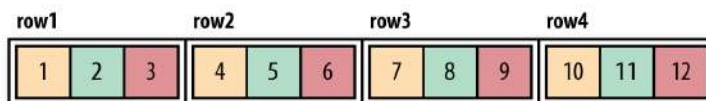
A column-oriented layout permits columns that are not accessed in a query to be skipped. Consider a query of the table in [Figure 5-4](#) that processes only column 2. With row-oriented storage, like a sequence file, the whole row (stored in a sequence file record) is loaded into memory, even though only the second column is actually read. Lazy deserialization saves some processing cycles by deserializing only the column fields that are accessed, but it can't avoid the cost of reading each row's bytes from disk.

With column-oriented storage, only the column 2 parts of the file (highlighted in the figure) need to be read into memory. In general, column-oriented formats work well when queries access only a small number of columns in the table. Conversely, row-oriented formats are appropriate when a large number of columns of a single row are needed for processing at the same time.

Logical table

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

Row-oriented layout (SequenceFile)



Column-oriented layout (RCFile)

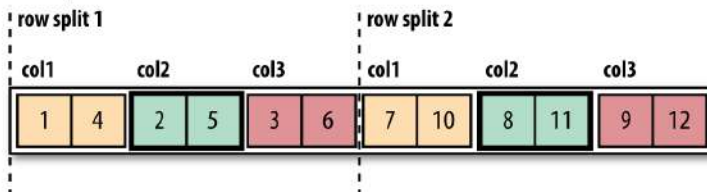


Figure 5-4. Row-oriented versus column-oriented storage

Column-oriented formats need more memory for reading and writing, since they have to buffer a row split in memory, rather than just a single row. Also, it's not usually possible to control when writes occur (via flush or sync operations), so column-oriented formats are not suited to streaming writes, as the current file cannot be recovered if the writer process fails. On the other hand, row-oriented formats like sequence files and Avro datafiles can be read up to the last sync point after a writer failure. It is for this reason that Flume (see [Chapter 14](#)) uses row-oriented formats.

The first column-oriented file format in Hadoop was Hive's *RCFile*, short for *Record Columnar File*. It has since been superseded by Hive's *ORCFile* (*Optimized Record Columnar File*), and *Parquet* (covered in [Chapter 13](#)). Parquet is a general-purpose column-oriented file format based on Google's Dremel, and has wide support across Hadoop components. Avro also has a column-oriented format called *Trevni*.

PART II

MapReduce

Developing a MapReduce Application

In [Chapter 2](#), we introduced the MapReduce model. In this chapter, we look at the practical aspects of developing a MapReduce application in Hadoop.

Writing a program in MapReduce follows a certain pattern. You start by writing your map and reduce functions, ideally with unit tests to make sure they do what you expect. Then you write a driver program to run a job, which can run from your IDE using a small subset of the data to check that it is working. If it fails, you can use your IDE's debugger to find the source of the problem. With this information, you can expand your unit tests to cover this case and improve your mapper or reducer as appropriate to handle such input correctly.

When the program runs as expected against the small dataset, you are ready to unleash it on a cluster. Running against the full dataset is likely to expose some more issues, which you can fix as before, by expanding your tests and altering your mapper or reducer to handle the new cases. Debugging failing programs in the cluster is a challenge, so we'll look at some common techniques to make it easier.

After the program is working, you may wish to do some tuning, first by running through some standard checks for making MapReduce programs faster and then by doing task profiling. Profiling distributed programs is not easy, but Hadoop has hooks to aid in the process.

Before we start writing a MapReduce program, however, we need to set up and configure the development environment. And to do that, we need to learn a bit about how Hadoop does configuration.

The Configuration API

Components in Hadoop are configured using Hadoop's own configuration API. An instance of the `Configuration` class (found in the `org.apache.hadoop.conf` package)

represents a collection of configuration *properties* and their values. Each property is named by a `String`, and the type of a value may be one of several, including Java primitives such as `boolean`, `int`, `long`, and `float`; other useful types such as `String`, `Class`, and `java.io.File`; and collections of `Strings`.

Configurations read their properties from *resources*—XML files with a simple structure for defining name-value pairs. See [Example 6-1](#).

Example 6-1. A simple configuration file, configuration-1.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>color</name>
    <value>yellow</value>
    <description>Color</description>
  </property>

  <property>
    <name>size</name>
    <value>10</value>
    <description>Size</description>
  </property>

  <property>
    <name>weight</name>
    <value>heavy</value>
    <final>true</final>
    <description>Weight</description>
  </property>

  <property>
    <name>size-weight</name>
    <value>${size},${weight}</value>
    <description>Size and weight</description>
  </property>
</configuration>
```

Assuming this `Configuration` is in a file called *configuration-1.xml*, we can access its properties using a piece of code like this:

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
assertThat(conf.get("color"), is("yellow"));
assertThat(conf.getInt("size", 0), is(10));
assertThat(conf.get("breadth", "wide"), is("wide"));
```

There are a couple of things to note: type information is not stored in the XML file; instead, properties can be interpreted as a given type when they are read. Also, the `get()` methods allow you to specify a default value, which is used if the property is not defined in the XML file, as in the case of `breadth` here.

Combining Resources

Things get interesting when more than one resource is used to define a Configuration. This is used in Hadoop to separate out the default properties for the system, defined internally in a file called *core-default.xml*, from the site-specific overrides in *core-site.xml*. The file in [Example 6-2](#) defines the size and weight properties.

Example 6-2. A second configuration file, configuration-2.xml

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>size</name>
    <value>12</value>
  </property>

  <property>
    <name>weight</name>
    <value>light</value>
  </property>
</configuration>
```

Resources are added to a Configuration in order:

```
Configuration conf = new Configuration();
conf.addResource("configuration-1.xml");
conf.addResource("configuration-2.xml");
```

Properties defined in resources that are added later override the earlier definitions. So the size property takes its value from the second configuration file, *configuration-2.xml*:

```
assertThat(conf.getInt("size", 0), is(12));
```

However, properties that are marked as `final` cannot be overridden in later definitions. The weight property is `final` in the first configuration file, so the attempt to override it in the second fails, and it takes the value from the first:

```
assertThat(conf.get("weight"), is("heavy"));
```

Attempting to override `final` properties usually indicates a configuration error, so this results in a warning message being logged to aid diagnosis. Administrators mark properties as `final` in the daemon's site files that they don't want users to change in their client-side configuration files or job submission parameters.

Variable Expansion

Configuration properties can be defined in terms of other properties, or system properties. For example, the property `size-weight` in the first configuration file is defined as `${size}.${weight}`, and these properties are expanded using the values found in the configuration:

```
assertThat(conf.get("size-weight"), is("12,heavy"));
```

System properties take priority over properties defined in resource files:

```
System.setProperty("size", "14");  
assertThat(conf.get("size-weight"), is("14,heavy"));
```

This feature is useful for overriding properties on the command line by using `-Dproperty=value` JVM arguments.

Note that although configuration properties can be defined in terms of system properties, unless system properties are redefined using configuration properties, they are *not* accessible through the configuration API. Hence:

```
System.setProperty("length", "2");  
assertThat(conf.get("length"), is((String) null));
```

Setting Up the Development Environment

The first step is to create a project so you can build MapReduce programs and run them in local (standalone) mode from the command line or within your IDE. The Maven Project Object Model (POM) in [Example 6-3](#) shows the dependencies needed for building and testing MapReduce programs.

Example 6-3. A Maven POM for building and testing a MapReduce application

```
<project>  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>com.hadoopbook</groupId>  
  <artifactId>hadoop-book-mr-dev</artifactId>  
  <version>4.0</version>  
  <properties>  
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
    <hadoop.version>2.5.1</hadoop.version>  
  </properties>  
  <dependencies>  
    <!-- Hadoop main client artifact -->  
    <dependency>  
      <groupId>org.apache.hadoop</groupId>  
      <artifactId>hadoop-client</artifactId>  
      <version>${hadoop.version}</version>  
    </dependency>  
    <!-- Unit test artifacts -->  
    <dependency>  
      <groupId>junit</groupId>  
      <artifactId>junit</artifactId>  
      <version>4.11</version>  
      <scope>test</scope>  
    </dependency>  
    <dependency>  
      <groupId>org.apache.mrunit</groupId>  
      <artifactId>mrunit</artifactId>
```

```

    <version>1.1.0</version>
    <classifier>hadoop2</classifier>
    <scope>test</scope>
  </dependency>
  <!-- Hadoop test artifact for running mini clusters -->
  <dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-miniclustert</artifactId>
    <version>${hadoop.version}</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<build>
  <finalName>hadoop-examples</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>2.5</version>
      <configuration>
        <outputDirectory>${basedir}</outputDirectory>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

The dependencies section is the interesting part of the POM. (It is straightforward to use another build tool, such as Gradle or Ant with Ivy, as long as you use the same set of dependencies defined here.) For building MapReduce jobs, you only need to have the `hadoop-client` dependency, which contains all the Hadoop client-side classes needed to interact with HDFS and MapReduce. For running unit tests, we use `junit`, and for writing MapReduce tests, we use `mrunit`. The `hadoop-miniclustert` library contains the “mini-” clusters that are useful for testing with Hadoop clusters running in a single JVM.

Many IDEs can read Maven POMs directly, so you can just point them at the directory containing the *pom.xml* file and start writing code. Alternatively, you can use Maven to generate configuration files for your IDE. For example, the following creates Eclipse configuration files so you can import the project into Eclipse:

```
% mvn eclipse:eclipse -DdownloadSources=true -DdownloadJavadocs=true
```

Managing Configuration

When developing Hadoop applications, it is common to switch between running the application locally and running it on a cluster. In fact, you may have several clusters you work with, or you may have a local “pseudodistributed” cluster that you like to test on (a pseudodistributed cluster is one whose daemons all run on the local machine; setting up this mode is covered in [Appendix A](#)).

One way to accommodate these variations is to have Hadoop configuration files containing the connection settings for each cluster you run against and specify which one you are using when you run Hadoop applications or tools. As a matter of best practice, it’s recommended to keep these files outside Hadoop’s installation directory tree, as this makes it easy to switch between Hadoop versions without duplicating or losing settings.

For the purposes of this book, we assume the existence of a directory called *conf* that contains three configuration files: *hadoop-local.xml*, *hadoop-localhost.xml*, and *hadoop-cluster.xml* (these are available in the example code for this book). Note that there is nothing special about the names of these files; they are just convenient ways to package up some configuration settings. (Compare this to [Table A-1](#) in [Appendix A](#), which sets out the equivalent server-side configurations.)

The *hadoop-local.xml* file contains the default Hadoop configuration for the default filesystem and the local (in-JVM) framework for running MapReduce jobs:

```
<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>file:///</value>
  </property>

  <property>
    <name>mapreduce.framework.name</name>
    <value>local</value>
  </property>

</configuration>
```

The settings in *hadoop-localhost.xml* point to a namenode and a YARN resource manager both running on localhost:

```
<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost/</value>
```

```

</property>

<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>

<property>
  <name>yarn.resourcemanager.address</name>
  <value>localhost:8032</value>
</property>

</configuration>

```

Finally, *hadoop-cluster.xml* contains details of the cluster’s namenode and YARN resource manager addresses (in practice, you would name the file after the name of the cluster, rather than “cluster” as we have here):

```

<?xml version="1.0"?>
<configuration>

  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://namenode/</value>
  </property>

  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>

  <property>
    <name>yarn.resourcemanager.address</name>
    <value>resourcemanager:8032</value>
  </property>

</configuration>

```

You can add other configuration properties to these files as needed.

Setting User Identity

The user identity that Hadoop uses for permissions in HDFS is determined by running the `whoami` command on the client system. Similarly, the group names are derived from the output of running `groups`.

If, however, your Hadoop user identity is different from the name of your user account on your client machine, you can explicitly set your Hadoop username by setting the `HADOOP_USER_NAME` environment variable. You can also override user group mappings by means of the `hadoop.user.group.static.mapping.overrides` configuration

property. For example, `dr.who=;preston=directors,inventors` means that the `dr.who` user is in no groups, but `preston` is in the `directors` and `inventors` groups.

You can set the user identity that the Hadoop web interfaces run as by setting the `hadoop.http.staticuser.user` property. By default, it is `dr.who`, which is not a superuser, so system files are not accessible through the web interface.

Notice that, by default, there is no authentication with this system. See “Security” on page 309 for how to use Kerberos authentication with Hadoop.

With this setup, it is easy to use any configuration with the `-conf` command-line switch. For example, the following command shows a directory listing on the HDFS server running in pseudodistributed mode on `localhost`:

```
% hadoop fs -conf conf/hadoop-localhost.xml -ls .
Found 2 items
drwxr-xr-x - tom supergroup          0 2014-09-08 10:19 input
drwxr-xr-x - tom supergroup          0 2014-09-08 10:19 output
```

If you omit the `-conf` option, you pick up the Hadoop configuration in the `etc/hadoop` subdirectory under `$HADOOP_HOME`. Or, if `HADOOP_CONF_DIR` is set, Hadoop configuration files will be read from that location.



Here’s an alternative way of managing configuration settings. Copy the `etc/hadoop` directory from your Hadoop installation to another location, place the `*-site.xml` configuration files there (with appropriate settings), and set the `HADOOP_CONF_DIR` environment variable to the alternative location. The main advantage of this approach is that you don’t need to specify `-conf` for every command. It also allows you to isolate changes to files other than the Hadoop XML configuration files (e.g., `log4j.properties`) since the `HADOOP_CONF_DIR` directory has a copy of all the configuration files (see “Hadoop Configuration” on page 292).

Tools that come with Hadoop support the `-conf` option, but it’s straightforward to make your programs (such as programs that run MapReduce jobs) support it, too, using the `Tool` interface.

GenericOptionsParser, Tool, and ToolRunner

Hadoop comes with a few helper classes for making it easier to run jobs from the command line. `GenericOptionsParser` is a class that interprets common Hadoop command-line options and sets them on a `Configuration` object for your application to use as desired. You don’t usually use `GenericOptionsParser` directly, as it’s more

convenient to implement the Tool interface and run your application with the ToolRunner, which uses GenericOptionsParser internally:

```
public interface Tool extends Configurable {  
    int run(String [] args) throws Exception;  
}
```

Example 6-4 shows a very simple implementation of Tool that prints the keys and values of all the properties in the Tool's Configuration object.

Example 6-4. An example Tool implementation for printing the properties in a Configuration

```
public class ConfigurationPrinter extends Configured implements Tool {  
  
    static {  
        Configuration.addDefaultResource("hdfs-default.xml");  
        Configuration.addDefaultResource("hdfs-site.xml");  
        Configuration.addDefaultResource("yarn-default.xml");  
        Configuration.addDefaultResource("yarn-site.xml");  
        Configuration.addDefaultResource("mapred-default.xml");  
        Configuration.addDefaultResource("mapred-site.xml");  
    }  
  
    @Override  
    public int run(String[] args) throws Exception {  
        Configuration conf = getConf();  
        for (Entry<String, String> entry: conf) {  
            System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());  
        }  
        return 0;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new ConfigurationPrinter(), args);  
        System.exit(exitCode);  
    }  
}
```

We make ConfigurationPrinter a subclass of Configured, which is an implementation of the Configurable interface. All implementations of Tool need to implement Configurable (since Tool extends it), and subclassing Configured is often the easiest way to achieve this. The run() method obtains the Configuration using Configuration's getConf() method and then iterates over it, printing each property to standard output.

The static block makes sure that the HDFS, YARN, and MapReduce configurations are picked up, in addition to the core ones (which Configuration knows about already).

ConfigurationPrinter's main() method does not invoke its own run() method directly. Instead, we call ToolRunner's static run() method, which takes care of creating

a Configuration object for the Tool before calling its `run()` method. ToolRunner also uses a `GenericOptionsParser` to pick up any standard options specified on the command line and to set them on the Configuration instance. We can see the effect of picking up the properties specified in *conf/hadoop-localhost.xml* by running the following commands:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop ConfigurationPrinter -conf conf/hadoop-localhost.xml \
  | grep yarn.resourcemanager.address=
yarn.resourcemanager.address=localhost:8032
```

Which Properties Can I Set?

`ConfigurationPrinter` is a useful tool for discovering what a property is set to in your environment. For a running daemon, like the namenode, you can see its configuration by viewing the */conf* page on its web server. (See [Table 10-6](#) to find port numbers.)

You can also see the default settings for all the public properties in Hadoop by looking in the *share/doc* directory of your Hadoop installation for files called *core-default.xml*, *hdfs-default.xml*, *yarn-default.xml*, and *mapred-default.xml*. Each property has a description that explains what it is for and what values it can be set to.

The default settings files' documentation can be found online at pages linked from <http://hadoop.apache.org/docs/current/> (look for the “Configuration” heading in the navigation). You can find the defaults for a particular Hadoop release by replacing *current* in the preceding URL with *r<version>*—for example, <http://hadoop.apache.org/docs/r2.5.0/>.

Be aware that some properties have no effect when set in the client configuration. For example, if you set `yarn.nodemanager.resource.memory-mb` in your job submission with the expectation that it would change the amount of memory available to the node managers running your job, you would be disappointed, because this property is honored only if set in the node manager's *yarn-site.xml* file. In general, you can tell the component where a property should be set by its name, so the fact that `yarn.nodemanager.resource.memory-mb` starts with `yarn.nodemanager` gives you a clue that it can be set only for the node manager daemon. This is not a hard and fast rule, however, so in some cases you may need to resort to trial and error, or even to reading the source.

Configuration property names have changed in Hadoop 2 onward, in order to give them a more regular naming structure. For example, the HDFS properties pertaining to the namenode have been changed to have a `dfs.namenode` prefix, so `dfs.name.dir` is now `dfs.namenode.name.dir`. Similarly, MapReduce properties have the `mapreduce` prefix rather than the older `mapred` prefix, so `mapred.job.name` is now `mapreduce.job.name`.

This book uses the new property names to avoid deprecation warnings. The old property names still work, however, and they are often referred to in older documentation. You can find a table listing the deprecated property names and their replacements on the [Hadoop website](#).

We discuss many of Hadoop's most important configuration properties throughout this book.

GenericOptionsParser also allows you to set individual properties. For example:

```
% hadoop ConfigurationPrinter -D color=yellow | grep color  
color=yellow
```

Here, the `-D` option is used to set the configuration property with key `color` to the value `yellow`. Options specified with `-D` take priority over properties from the configuration files. This is very useful because you can put defaults into configuration files and then override them with the `-D` option as needed. A common example of this is setting the number of reducers for a MapReduce job via `-D mapreduce.job.reduces=n`. This will override the number of reducers set on the cluster or set in any client-side configuration files.

The other options that GenericOptionsParser and ToolRunner support are listed in [Table 6-1](#). You can find more on Hadoop's configuration API in [“The Configuration API” on page 141](#).



Do not confuse setting Hadoop properties using the `-D property=value` option to GenericOptionsParser (and ToolRunner) with setting JVM system properties using the `-Dproperty=value` option to the `java` command. The syntax for JVM system properties does not allow any whitespace between the `D` and the property name, whereas GenericOptionsParser does allow whitespace.

JVM system properties are retrieved from the `java.lang.System` class, but Hadoop properties are accessible only from a `Configuration` object. So, the following command will print nothing, even though the `color` system property has been set (via `HADOOP_OPTS`), because the `System` class is not used by `ConfigurationPrinter`:

```
% HADOOP_OPTS='-Dcolor=yellow' \  
hadoop ConfigurationPrinter | grep color
```

If you want to be able to set configuration through system properties, you need to mirror the system properties of interest in the configuration file. See [“Variable Expansion” on page 143](#) for further discussion.

Table 6-1. *GenericOptionsParser and ToolRunner options*

Option	Description
<code>-D <i>property=value</i></code>	Sets the given Hadoop configuration property to the given value. Overrides any default or site properties in the configuration and any properties set via the <code>-conf</code> option.
<code>-conf <i>filename ...</i></code>	Adds the given files to the list of resources in the configuration. This is a convenient way to set site properties or to set a number of properties at once.
<code>-fs <i>uri</i></code>	Sets the default filesystem to the given URI. Shortcut for <code>-D fs.defaultFS=<i>uri</i></code> .
<code>-jt <i>host:port</i></code>	Sets the YARN resource manager to the given host and port. (In Hadoop 1, it sets the jobtracker address, hence the option name.) Shortcut for <code>-D yarn.resource.manager.address=<i>host:port</i></code> .
<code>-files <i>file1,file2,...</i></code>	Copies the specified files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and makes them available to MapReduce programs in the task's working directory. (See “Distributed Cache” on page 274 for more on the distributed cache mechanism for copying files to machines in the cluster.)
<code>-archives <i>archive1,archive2,...</i></code>	Copies the specified archives from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS), unarchives them, and makes them available to MapReduce programs in the task's working directory.
<code>-libjars <i>jar1,jar2,...</i></code>	Copies the specified JAR files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and adds them to the MapReduce task's classpath. This option is a useful way of shipping JAR files that a job is dependent on.

Writing a Unit Test with MRUnit

The map and reduce functions in MapReduce are easy to test in isolation, which is a consequence of their functional style. **MRUnit** is a testing library that makes it easy to pass known inputs to a mapper or a reducer and check that the outputs are as expected. MRUnit is used in conjunction with a standard test execution framework, such as JUnit, so you can run the tests for MapReduce jobs in your normal development environment. For example, all of the tests described here can be run from within an IDE by following the instructions in [“Setting Up the Development Environment” on page 144](#).

Mapper

The test for the mapper is shown in [Example 6-5](#).

Example 6-5. Unit test for MaxTemperatureMapper

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.MapperDriver;
import org.junit.*;

public class MaxTemperatureMapperTest {

    @Test
    public void processesValidRecord() throws IOException, InterruptedException {
        Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
                               // Year ^^^^
                               "99999V0203201N00261220001CN9999999N9-00111+9999999999");
                               // Temperature ^^^^^
        new MapperDriver<LongWritable, Text, Text, IntWritable>()
            .withMapper(new MaxTemperatureMapper())
            .withInput(new LongWritable(0), value)
            .withOutput(new Text("1950"), new IntWritable(-11))
            .runTest();
    }
}
```

The idea of the test is very simple: pass a weather record as input to the mapper, and check that the output is the year and temperature reading.

Since we are testing the mapper, we use MRUnit's MapDriver, which we configure with the mapper under test (MaxTemperatureMapper), the input key and value, and the expected output key (a Text object representing the year, 1950) and expected output value (an IntWritable representing the temperature, -1.1°C), before finally calling the runTest() method to execute the test. If the expected output values are not emitted by the mapper, MRUnit will fail the test. Notice that the input key could be set to any value because our mapper ignores it.

Proceeding in a test-driven fashion, we create a Mapper implementation that passes the test (see [Example 6-6](#)). Because we will be evolving the classes in this chapter, each is put in a different package indicating its version for ease of exposition. For example, v1.MaxTemperatureMapper is version 1 of MaxTemperatureMapper. In reality, of course, you would evolve classes without repackaging them.

Example 6-6. First version of a Mapper that passes MaxTemperatureMapperTest

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

    @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 = Integer.parseInt(line.substring(87, 92));
    context.write(new Text(year), new IntWritable(airTemperature));
}
}

```

This is a very simple implementation that pulls the year and temperature fields from the line and writes them to the Context. Let's add a test for missing values, which in the raw data are represented by a temperature of +9999:

```

@Test
public void ignoresMissingTemperatureRecord() throws IOException,
    InterruptedException {
    Text value = new Text("0043011990999991950051518004+68750+023550FM-12+0382" +
        // Year ^^^^
        "99999V0203201N00261220001CN9999999N9+99991+9999999999");
    // Temperature ^^^^^
    new MapDriver<LongWritable, Text, Text, IntWritable>()
        .withMapper(new MaxTemperatureMapper())
        .withInput(new LongWritable(0), value)
        .runTest();
}

```

A MapDriver can be used to check for zero, one, or more output records, according to the number of times that `withOutput()` is called. In our application, since records with missing temperatures should be filtered out, this test asserts that no output is produced for this particular input value.

The new test fails since +9999 is not treated as a special case. Rather than putting more logic into the mapper, it makes sense to factor out a parser class to encapsulate the parsing logic; see [Example 6-7](#).

Example 6-7. A class for parsing weather records in NCDC format

```

public class NcdcRecordParser {

    private static final int MISSING_TEMPERATURE = 9999;

    private String year;
    private int airTemperature;
    private String quality;

    public void parse(String record) {
        year = record.substring(15, 19);
        String airTemperatureString;
        // Remove leading plus sign as parseInt doesn't like them (pre-Java 7)
        if (record.charAt(87) == '+') {
            airTemperatureString = record.substring(88, 92);

```

```

    } else {
        airTemperatureString = record.substring(87, 92);
    }
    airTemperature = Integer.parseInt(airTemperatureString);
    quality = record.substring(92, 93);
}

public void parse(Text record) {
    parse(record.toString());
}

public boolean isValidTemperature() {
    return airTemperature != MISSING_TEMPERATURE && quality.matches("[01459]");
}

public String getYear() {
    return year;
}

public int getAirTemperature() {
    return airTemperature;
}
}

```

The resulting mapper (version 2) is much simpler (see [Example 6-8](#)). It just calls the parser's `parse()` method, which parses the fields of interest from a line of input, checks whether a valid temperature was found using the `isValidTemperature()` query method, and, if it was, retrieves the year and the temperature using the getter methods on the parser. Notice that we check the quality status field as well as checking for missing temperatures in `isValidTemperature()`, to filter out poor temperature readings.



Another benefit of creating a parser class is that it makes it easy to write related mappers for similar jobs without duplicating code. It also gives us the opportunity to write unit tests directly against the parser, for more targeted testing.

Example 6-8. A Mapper that uses a utility class to parse records

```

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

    private NcdcRecordParser parser = new NcdcRecordParser();

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

        parser.parse(value);
        if (parser.isValidTemperature()) {

```

```

        context.write(new Text(parser.getYear()),
            new IntWritable(parser.getAirTemperature()));
    }
}
}

```

With the tests for the mapper now passing, we move on to writing the reducer.

Reducer

The reducer has to find the maximum value for a given key. Here's a simple test for this feature, which uses a `ReduceDriver`:

```

@Test
public void returnsMaximumIntegerInValues() throws IOException,
    InterruptedException {
    new ReduceDriver<Text, IntWritable, Text, IntWritable>()
        .withReducer(new MaxTemperatureReducer())
        .withInput(new Text("1950"),
            Arrays.asList(new IntWritable(10), new IntWritable(5)))
        .withOutput(new Text("1950"), new IntWritable(10))
        .runTest();
}

```

We construct a list of some `IntWritable` values and then verify that `MaxTemperatureReducer` picks the largest. The code in [Example 6-9](#) is for an implementation of `MaxTemperatureReducer` that passes the test.

Example 6-9. Reducer for the maximum temperature example

```

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));
    }
}

```

Running Locally on Test Data

Now that we have the mapper and reducer working on controlled inputs, the next step is to write a job driver and run it on some test data on a development machine.

Running a Job in a Local Job Runner

Using the Tool interface introduced earlier in the chapter, it's easy to write a driver to run our MapReduce job for finding the maximum temperature by year (see `MaxTemperatureDriver` in [Example 6-10](#)).

Example 6-10. Application to find the maximum temperature

```
public class MaxTemperatureDriver extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s [generic options] <input> <output>\n",
                               getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.err);
            return -1;
        }

        Job job = new Job(getConf(), "Max temperature");
        job.setJarByClass(getClass());

        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);
        job.setOutputValueClass(IntWritable.class);

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

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

`MaxTemperatureDriver` implements the `Tool` interface, so we get the benefit of being able to set the options that `GenericOptionsParser` supports. The `run()` method constructs a `Job` object based on the tool's configuration, which it uses to launch a job. Among the possible job configuration parameters, we set the input and output file paths; the mapper, reducer, and combiner classes; and the output types (the input types are determined by the input format, which defaults to `TextInputFormat` and has `LongWritable` keys and `Text` values). It's also a good idea to set a name for the job (`Max temperature`) so that you can pick it out in the job list during execution and after it has

completed. By default, the name is the name of the JAR file, which normally is not particularly descriptive.

Now we can run this application against some local files. Hadoop comes with a local job runner, a cut-down version of the MapReduce execution engine for running MapReduce jobs in a single JVM. It's designed for testing and is very convenient for use in an IDE, since you can run it in a debugger to step through the code in your mapper and reducer.

The local job runner is used if `mapreduce.framework.name` is set to `local`, which is the default.¹

From the command line, we can run the driver by typing:

```
% mvn compile
% export HADOOP_CLASSPATH=target/classes/
% hadoop v2.MaxTemperatureDriver -conf conf/hadoop-local.xml \
  input/ncdc/micro output
```

Equivalently, we could use the `-fs` and `-jt` options provided by `GenericOptionsParser`:

```
% hadoop v2.MaxTemperatureDriver -fs file:/// -jt local input/ncdc/micro output
```

This command executes `MaxTemperatureDriver` using input from the local `input/ncdc/micro` directory, producing output in the local `output` directory. Note that although we've set `-fs` so we use the local filesystem (`file:///`), the local job runner will actually work fine against any filesystem, including HDFS (and it can be handy to do this if you have a few files that are on HDFS).

We can examine the output on the local filesystem:

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

Testing the Driver

Apart from the flexible configuration options offered by making your application implement `Tool`, you also make it more testable because it allows you to inject an arbitrary `Configuration`. You can take advantage of this to write a test that uses a local job runner to run a job against known input data, which checks that the output is as expected.

There are two approaches to doing this. The first is to use the local job runner and run the job against a test file on the local filesystem. The code in [Example 6-11](#) gives an idea of how to do this.

1. In Hadoop 1, `mapred.job.tracker` determines the means of execution: `local` for the local job runner, or a colon-separated host and port pair for a jobtracker address.

Example 6-11. A test for `MaxTemperatureDriver` that uses a local, in-process job runner

```
@Test
public void test() throws Exception {
    Configuration conf = new Configuration();
    conf.set("fs.defaultFS", "file:///");
    conf.set("mapreduce.framework.name", "local");
    conf.setInt("mapreduce.task.io.sort.mb", 1);

    Path input = new Path("input/ncdc/micro");
    Path output = new Path("output");

    FileSystem fs = FileSystem.getLocal(conf);
    fs.delete(output, true); // delete old output

    MaxTemperatureDriver driver = new MaxTemperatureDriver();
    driver.setConf(conf);

    int exitCode = driver.run(new String[] {
        input.toString(), output.toString() });
    assertEquals(exitCode, 0);

    checkOutput(conf, output);
}
```

The test explicitly sets `fs.defaultFS` and `mapreduce.framework.name` so it uses the local filesystem and the local job runner. It then runs the `MaxTemperatureDriver` via its `Tool` interface against a small amount of known data. At the end of the test, the `checkOutput()` method is called to compare the actual output with the expected output, line by line.

The second way of testing the driver is to run it using a “mini-” cluster. Hadoop has a set of testing classes, called `MiniDFSCluster`, `MiniMRCluster`, and `MiniYARNCluster`, that provide a programmatic way of creating in-process clusters. Unlike the local job runner, these allow testing against the full HDFS, MapReduce, and YARN machinery. Bear in mind, too, that node managers in a mini-cluster launch separate JVMs to run tasks in, which can make debugging more difficult.



You can run a mini-cluster from the command line too, with the following:

```
% hadoop jar \  
  $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-tests.jar \  
  minicluster
```

Mini-clusters are used extensively in Hadoop’s own automated test suite, but they can be used for testing user code, too. Hadoop’s `ClusterMapReduceTestCase` abstract class provides a useful base for writing such a test, handles the details of starting and stopping

the in-process HDFS and YARN clusters in its `setUp()` and `tearDown()` methods, and generates a suitable `Configuration` object that is set up to work with them. Subclasses need only populate data in HDFS (perhaps by copying from a local file), run a MapReduce job, and confirm the output is as expected. Refer to the `MaxTemperatureDriver MiniTest` class in the example code that comes with this book for the listing.

Tests like this serve as regression tests, and are a useful repository of input edge cases and their expected results. As you encounter more test cases, you can simply add them to the input file and update the file of expected output accordingly.

Running on a Cluster

Now that we are happy with the program running on a small test dataset, we are ready to try it on the full dataset on a Hadoop cluster. [Chapter 10](#) covers how to set up a fully distributed cluster, although you can also work through this section on a pseudo-distributed cluster.

Packaging a Job

The local job runner uses a single JVM to run a job, so as long as all the classes that your job needs are on its classpath, then things will just work.

In a distributed setting, things are a little more complex. For a start, a job's classes must be packaged into a *job JAR file* to send to the cluster. Hadoop will find the job JAR automatically by searching for the JAR on the driver's classpath that contains the class set in the `setJarByClass()` method (on `JobConf` or `Job`). Alternatively, if you want to set an explicit JAR file by its file path, you can use the `setJar()` method. (The JAR file path may be local or an HDFS file path.)

Creating a job JAR file is conveniently achieved using a build tool such as Ant or Maven. Given the POM in [Example 6-3](#), the following Maven command will create a JAR file called *hadoop-examples.jar* in the project directory containing all of the compiled classes:

```
% mvn package -DskipTests
```

If you have a single job per JAR, you can specify the main class to run in the JAR file's manifest. If the main class is not in the manifest, it must be specified on the command line (as we will see shortly when we run the job).

Any dependent JAR files can be packaged in a *lib* subdirectory in the job JAR file, although there are other ways to include dependencies, discussed later. Similarly, resource files can be packaged in a *classes* subdirectory. (This is analogous to a Java *Web application archive*, or WAR, file, except in that case the JAR files go in a *WEB-INF/lib* subdirectory and classes go in a *WEB-INF/classes* subdirectory in the WAR file.)

The client classpath

The user's client-side classpath set by `hadoop jar <jar>` is made up of:

- The job JAR file
- Any JAR files in the *lib* directory of the job JAR file, and the *classes* directory (if present)
- The classpath defined by `HADOOP_CLASSPATH`, if set

Incidentally, this explains why you have to set `HADOOP_CLASSPATH` to point to dependent classes and libraries if you are running using the local job runner without a job JAR (`hadoop CLASSNAME`).

The task classpath

On a cluster (and this includes pseudodistributed mode), map and reduce tasks run in separate JVMs, and their classpaths are *not* controlled by `HADOOP_CLASSPATH`. `HADOOP_CLASSPATH` is a client-side setting and only sets the classpath for the driver JVM, which submits the job.

Instead, the user's task classpath is comprised of the following:

- The job JAR file
- Any JAR files contained in the *lib* directory of the job JAR file, and the *classes* directory (if present)
- Any files added to the distributed cache using the `-libjars` option (see [Table 6-1](#)), or the `addFileToClassPath()` method on `DistributedCache` (old API), or `Job` (new API)

Packaging dependencies

Given these different ways of controlling what is on the client and task classpaths, there are corresponding options for including library dependencies for a job:

- Unpack the libraries and repack them in the job JAR.
- Package the libraries in the *lib* directory of the job JAR.
- Keep the libraries separate from the job JAR, and add them to the client classpath via `HADOOP_CLASSPATH` and to the task classpath via `-libjars`.

The last option, using the distributed cache, is simplest from a build point of view because dependencies don't need rebundling in the job JAR. Also, using the distributed cache can mean fewer transfers of JAR files around the cluster, since files may be cached on a node between tasks. (You can read more about the distributed cache [on page 274](#).)

Task classpath precedence

User JAR files are added to the end of both the client classpath and the task classpath, which in some cases can cause a dependency conflict with Hadoop's built-in libraries if Hadoop uses a different, incompatible version of a library that your code uses. Sometimes you need to be able to control the task classpath order so that your classes are picked up first. On the client side, you can force Hadoop to put the user classpath first in the search order by setting the `HADOOP_USER_CLASSPATH_FIRST` environment variable to `true`. For the task classpath, you can set `mapreduce.job.user.classpath.first` to `true`. Note that by setting these options you change the class loading for Hadoop framework dependencies (but only in your job), which could potentially cause the job submission or task to fail, so use these options with caution.

Launching a Job

To launch the job, we need to run the driver, specifying the cluster that we want to run the job on with the `-conf` option (we equally could have used the `-fs` and `-jt` options):

```
% unset HADOOP_CLASSPATH
% hadoop jar hadoop-examples.jar v2.MaxTemperatureDriver \
  -conf conf/hadoop-cluster.xml input/ncdc/all max-temp
```



We unset the `HADOOP_CLASSPATH` environment variable because we don't have any third-party dependencies for this job. If it were left set to `target/classes/` (from earlier in the chapter), Hadoop wouldn't be able to find the job JAR; it would load the `MaxTemperatureDriver` class from `target/classes` rather than the JAR, and the job would fail.

The `waitForCompletion()` method on `Job` launches the job and polls for progress, writing a line summarizing the map and reduce's progress whenever either changes. Here's the output (some lines have been removed for clarity):

```
14/09/12 06:38:11 INFO input.FileInputFormat: Total input paths to process : 101
14/09/12 06:38:11 INFO impl.YarnClientImpl: Submitted application
application_1410450250506_0003
14/09/12 06:38:12 INFO mapreduce.Job: Running job: job_1410450250506_0003
14/09/12 06:38:26 INFO mapreduce.Job: map 0% reduce 0%
...
14/09/12 06:45:24 INFO mapreduce.Job: map 100% reduce 100%
14/09/12 06:45:24 INFO mapreduce.Job: Job job_1410450250506_0003 completed
successfully
14/09/12 06:45:24 INFO mapreduce.Job: Counters: 49
  File System Counters
    FILE: Number of bytes read=93995
    FILE: Number of bytes written=10273563
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
```

```

FILE: Number of write operations=0
HDFS: Number of bytes read=33485855415
HDFS: Number of bytes written=904
HDFS: Number of read operations=327
HDFS: Number of large read operations=0
HDFS: Number of write operations=16
Job Counters
  Launched map tasks=101
  Launched reduce tasks=8
  Data-local map tasks=101
  Total time spent by all maps in occupied slots (ms)=5954495
  Total time spent by all reduces in occupied slots (ms)=74934
  Total time spent by all map tasks (ms)=5954495
  Total time spent by all reduce tasks (ms)=74934
  Total vcore-seconds taken by all map tasks=5954495
  Total vcore-seconds taken by all reduce tasks=74934
  Total megabyte-seconds taken by all map tasks=6097402880
  Total megabyte-seconds taken by all reduce tasks=76732416
Map-Reduce Framework
  Map input records=1209901509
  Map output records=1143764653
  Map output bytes=10293881877
  Map output materialized bytes=14193
  Input split bytes=14140
  Combine input records=1143764772
  Combine output records=234
  Reduce input groups=100
  Reduce shuffle bytes=14193
  Reduce input records=115
  Reduce output records=100
  Spilled Records=379
  Shuffled Maps =808
  Failed Shuffles=0
  Merged Map outputs=808
  GC time elapsed (ms)=101080
  CPU time spent (ms)=5113180
  Physical memory (bytes) snapshot=60509106176
  Virtual memory (bytes) snapshot=167657209856
  Total committed heap usage (bytes)=68220878848
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=33485841275
File Output Format Counters
  Bytes Written=90

```

The output includes more useful information. Before the job starts, its ID is printed; this is needed whenever you want to refer to the job—in logfiles, for example—or when interrogating it via the `mapred job` command. When the job is complete, its statistics (known as counters) are printed out. These are very useful for confirming that the job did what you expected. For example, for this job, we can see that 1.2 billion records were analyzed (“Map input records”), read from around 34 GB of compressed files on HDFS (“HDFS: Number of bytes read”). The input was broken into 101 gzipped files of reasonable size, so there was no problem with not being able to split them.

You can find out more about what the counters mean in [“Built-in Counters” on page 247](#).

Job, Task, and Task Attempt IDs

In Hadoop 2, MapReduce job IDs are generated from YARN application IDs that are created by the YARN resource manager. The format of an application ID is composed of the time that the resource manager (not the application) started and an incrementing counter maintained by the resource manager to uniquely identify the application to that instance of the resource manager. So the application with this ID:

```
application_1410450250506_0003
```

is the third (0003; application IDs are 1-based) application run by the resource manager, which started at the time represented by the timestamp 1410450250506. The counter is formatted with leading zeros to make IDs sort nicely—in directory listings, for example. However, when the counter reaches 10000, it is *not* reset, resulting in longer application IDs (which don’t sort so well).

The corresponding job ID is created simply by replacing the application prefix of an application ID with a job prefix:

```
job_1410450250506_0003
```

Tasks belong to a job, and their IDs are formed by replacing the job prefix of a job ID with a task prefix and adding a suffix to identify the task within the job. For example:

```
task_1410450250506_0003_m_000003
```

is the fourth (000003; task IDs are 0-based) map (m) task of the job with ID `job_1410450250506_0003`. The task IDs are created for a job when it is initialized, so they do not necessarily dictate the order in which the tasks will be executed.

Tasks may be executed more than once, due to failure (see [“Task Failure” on page 193](#)) or speculative execution (see [“Speculative Execution” on page 204](#)), so to identify different instances of a task execution, task attempts are given unique IDs. For example:

```
attempt_1410450250506_0003_m_000003_0
```

is the first (0; attempt IDs are 0-based) attempt at running task task_1410450250506_0003_m_000003. Task attempts are allocated during the job run as needed, so their ordering represents the order in which they were created to run.

The MapReduce Web UI

Hadoop comes with a web UI for viewing information about your jobs. It is useful for following a job's progress while it is running, as well as finding job statistics and logs after the job has completed. You can find the UI at <http://resource-manager-host:8088/>.

The resource manager page

A screenshot of the home page is shown in [Figure 6-1](#). The “Cluster Metrics” section gives a summary of the cluster. This includes the number of applications currently running on the cluster (and in various other states), the number of resources available on the cluster (“Memory Total”), and information about node managers.

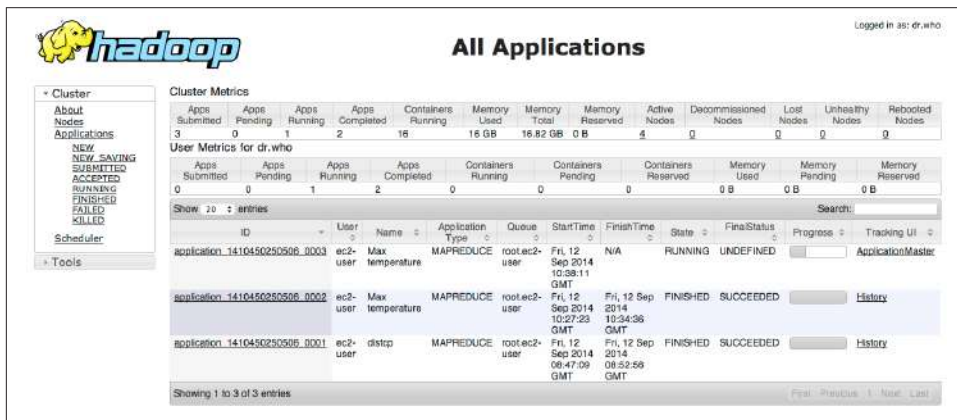


Figure 6-1. Screenshot of the resource manager page

The main table shows all the applications that have run or are currently running on the cluster. There is a search box that is useful for filtering the applications to find the ones you are interested in. The main view can show up to 100 entries per page, and the resource manager will keep up to 10,000 completed applications in memory at a time (set by `yarn.resourcemanager.max-completed-applications`), before they are only available from the job history page. Note also that the job history is persistent, so you can find jobs there from previous runs of the resource manager, too.

Job History

Job history refers to the events and configuration for a completed MapReduce job. It is retained regardless of whether the job was successful, in an attempt to provide useful information for the user running a job.

Job history files are stored in HDFS by the MapReduce application master, in a directory set by the `mapreduce.jobhistory.done-dir` property. Job history files are kept for one week before being deleted by the system.

The history log includes job, task, and attempt events, all of which are stored in a file in JSON format. The history for a particular job may be viewed through the web UI for the job history server (which is linked to from the resource manager page) or via the command line using `mapred job -history` (which you point at the job history file).

The MapReduce job page

Clicking on the link for the “Tracking UI” takes us to the application master’s web UI (or to the history page if the application has completed). In the case of MapReduce, this takes us to the job page, illustrated in [Figure 6-2](#).

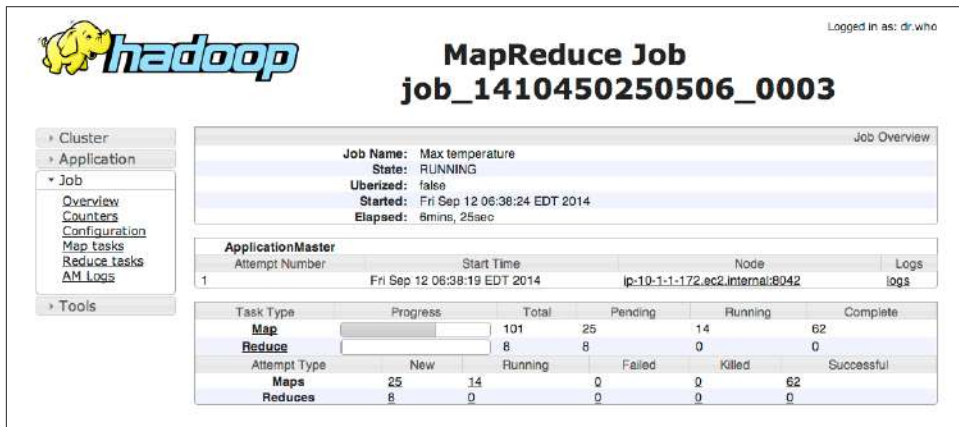


Figure 6-2. Screenshot of the job page

While the job is running, you can monitor its progress on this page. The table at the bottom shows the map progress and the reduce progress. “Total” shows the total number of map and reduce tasks for this job (a row for each). The other columns then show the state of these tasks: “Pending” (waiting to run), “Running,” or “Complete” (successfully run).

The lower part of the table shows the total number of failed and killed task attempts for the map or reduce tasks. Task attempts may be marked as killed if they are speculative execution duplicates, if the node they are running on dies, or if they are killed by a user. See [“Task Failure” on page 193](#) for background on task failure.

There also are a number of useful links in the navigation. For example, the “Configuration” link is to the consolidated configuration file for the job, containing all the properties and their values that were in effect during the job run. If you are unsure of what a particular property was set to, you can click through to inspect the file.

Retrieving the Results

Once the job is finished, there are various ways to retrieve the results. Each reducer produces one output file, so there are 30 part files named *part-r-00000* to *part-r-00029* in the *max-temp* directory.



As their names suggest, a good way to think of these “part” files is as parts of the *max-temp* “file.”

If the output is large (which it isn’t in this case), it is important to have multiple parts so that more than one reducer can work in parallel. Usually, if a file is in this partitioned form, it can still be used easily enough—as the input to another MapReduce job, for example. In some cases, you can exploit the structure of multiple partitions to do a map-side join, for example (see [“Map-Side Joins” on page 269](#)).

This job produces a very small amount of output, so it is convenient to copy it from HDFS to our development machine. The `-getmerge` option to the `hadoop fs` command is useful here, as it gets all the files in the directory specified in the source pattern and merges them into a single file on the local filesystem:

```
% hadoop fs -getmerge max-temp max-temp-local
% sort max-temp-local | tail
1991      607
1992      605
1993      567
1994      568
1995      567
1996      561
1997      565
1998      568
1999      568
2000      558
```

We sorted the output, as the reduce output partitions are unordered (owing to the hash partition function). Doing a bit of postprocessing of data from MapReduce is very

common, as is feeding it into analysis tools such as R, a spreadsheet, or even a relational database.

Another way of retrieving the output if it is small is to use the `-cat` option to print the output files to the console:

```
% hadoop fs -cat max-temp/*
```

On closer inspection, we see that some of the results don't look plausible. For instance, the maximum temperature for 1951 (not shown here) is 590°C! How do we find out what's causing this? Is it corrupt input data or a bug in the program?

Debugging a Job

The time-honored way of debugging programs is via print statements, and this is certainly possible in Hadoop. However, there are complications to consider: with programs running on tens, hundreds, or thousands of nodes, how do we find and examine the output of the debug statements, which may be scattered across these nodes? For this particular case, where we are looking for (what we think is) an unusual case, we can use a debug statement to log to standard error, in conjunction with updating the task's status message to prompt us to look in the error log. The web UI makes this easy, as we pass: [will see].

We also create a custom counter to count the total number of records with implausible temperatures in the whole dataset. This gives us valuable information about how to deal with the condition. If it turns out to be a common occurrence, we might need to learn more about the condition and how to extract the temperature in these cases, rather than simply dropping the records. In fact, when trying to debug a job, you should always ask yourself if you can use a counter to get the information you need to find out what's happening. Even if you need to use logging or a status message, it may be useful to use a counter to gauge the extent of the problem. (There is more on counters in “[Counters](#)” on page 247.)

If the amount of log data you produce in the course of debugging is large, you have a couple of options. One is to write the information to the map's output, rather than to standard error, for analysis and aggregation by the reduce task. This approach usually necessitates structural changes to your program, so start with the other technique first. The alternative is to write a program (in MapReduce, of course) to analyze the logs produced by your job.

We add our debugging to the mapper (version 3), as opposed to the reducer, as we want to find out what the source data causing the anomalous output looks like:

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

    enum Temperature {
```

```

        OVER_100
    }

    private NcdcRecordParser parser = new NcdcRecordParser();

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

        parser.parse(value);
        if (parser.isValidTemperature()) {
            int airTemperature = parser.getAirTemperature();
            if (airTemperature > 1000) {
                System.err.println("Temperature over 100 degrees for input: " + value);
                context.setStatus("Detected possibly corrupt record: see logs.");
                context.getCounter(Temperature.OVER_100).increment(1);
            }
            context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
        }
    }
}

```

If the temperature is over 100°C (represented by 1000, because temperatures are in tenths of a degree), we print a line to standard error with the suspect line, as well as updating the map's status message using the `setStatus()` method on `Context`, directing us to look in the log. We also increment a counter, which in Java is represented by a field of an enum type. In this program, we have defined a single field, `OVER_100`, as a way to count the number of records with a temperature of over 100°C.

With this modification, we recompile the code, re-create the JAR file, then rerun the job and, while it's running, go to the tasks page.

The tasks and task attempts pages

The job page has a number of links for viewing the tasks in a job in more detail. For example, clicking on the “Map” link brings us to a page that lists information for all of the map tasks. The screenshot in [Figure 6-3](#) shows this page for the job run with our debugging statements in the “Status” column for the task.

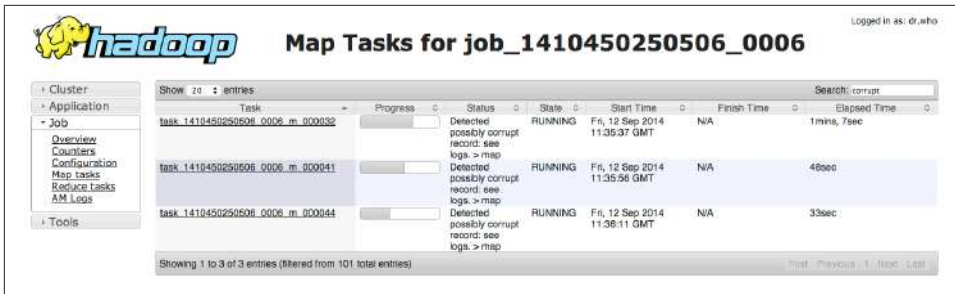


Figure 6-3. Screenshot of the tasks page

Clicking on the task link takes us to the task attempts page, which shows each task attempt for the task. Each task attempt page has links to the logfiles and counters. If we follow one of the links to the logfiles for the successful task attempt, we can find the suspect input record that we logged (the line is wrapped and truncated to fit on the page):

```
Temperature over 100 degrees for input:
0335999999433181957042302005+37950+139117SAO +0004RJSN V02011359003150070356999
999433201957010100005+35317+139650SAO +000899999V02002359002650076249N0040005...
```

This record seems to be in a different format from the others. For one thing, there are spaces in the line, which are not described in the specification.

When the job has finished, we can look at the value of the counter we defined to see how many records over 100°C there are in the whole dataset. Counters are accessible via the web UI or the command line:

```
% mapred job -counter job_1410450250506_0006 \
'v3.MaxTemperatureMapper$Temperature' OVER_100
3
```

The `-counter` option takes the job ID, counter group name (which is the fully qualified classname here), and counter name (the enum name). There are only three malformed records in the entire dataset of over a billion records. Throwing out bad records is standard for many big data problems, although we need to be careful in this case because we are looking for an extreme value—the maximum temperature rather than an aggregate measure. Still, throwing away three records is probably not going to change the result.

Handling malformed data

Capturing input data that causes a problem is valuable, as we can use it in a test to check that the mapper does the right thing. In this MRUnit test, we check that the counter is updated for the malformed input:

```

@Test
public void parsesMalformedTemperature() throws IOException,
    InterruptedException {
    Text value = new Text("0335999999433181957042302005+37950+139117SAO  +0004" +
        // Year ^^^^
        "RJSN V02011359003150070356999999433201957010100005+353");
    // Temperature ^^^^^
    Counters counters = new Counters();
    new MapDriver<LongWritable, Text, Text, IntWritable>()
        .withMapper(new MaxTemperatureMapper())
        .withInput(new LongWritable(0), value)
        .withCounters(counters)
        .runTest();
    Counter c = counters.findCounter(MaxTemperatureMapper.Temperature.MALFORMED);
    assertThat(c.getValue(), is(1L));
}

```

The record that was causing the problem is of a different format than the other lines we've seen. [Example 6-12](#) shows a modified program (version 4) using a parser that ignores each line with a temperature field that does not have a leading sign (plus or minus). We've also introduced a counter to measure the number of records that we are ignoring for this reason.

Example 6-12. Mapper for the maximum temperature example

```

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

    enum Temperature {
        MALFORMED
    }

    private NcdcRecordParser parser = new NcdcRecordParser();

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

        parser.parse(value);
        if (parser.isValidTemperature()) {
            int airTemperature = parser.getAirTemperature();
            context.write(new Text(parser.getYear()), new IntWritable(airTemperature));
        } else if (parser.isMalformedTemperature()) {
            System.err.println("Ignoring possibly corrupt input: " + value);
            context.getCounter(Temperature.MALFORMED).increment(1);
        }
    }
}

```

Hadoop Logs

Hadoop produces logs in various places, and for various audiences. These are summarized in [Table 6-2](#).

Table 6-2. Types of Hadoop logs

Logs	Primary audience	Description	Further information
System daemon logs	Administrators	Each Hadoop daemon produces a logfile (using log4j) and another file that combines standard out and error. Written in the directory defined by the HADOOP_LOG_DIR environment variable.	“System logfiles” on page 295 and “Logging” on page 330
HDFS audit logs	Administrators	A log of all HDFS requests, turned off by default. Written to the namenode’s log, although this is configurable.	“Audit Logging” on page 324
MapReduce job history logs	Users	A log of the events (such as task completion) that occur in the course of running a job. Saved centrally in HDFS.	“Job History” on page 166
MapReduce task logs	Users	Each task child process produces a logfile using log4j (called <i>syslog</i>), a file for data sent to standard out (<i>stdout</i>), and a file for standard error (<i>stderr</i>). Written in the <i>userlogs</i> subdirectory of the directory defined by the YARN_LOG_DIR environment variable.	This section

YARN has a service for *log aggregation* that takes the task logs for completed applications and moves them to HDFS, where they are stored in a container file for archival purposes. If this service is enabled (by setting `yarn.log-aggregation-enable` to `true` on the cluster), then task logs can be viewed by clicking on the *logs* link in the task attempt web UI, or by using the `mapred job -logs` command.

By default, log aggregation is not enabled. In this case, task logs can be retrieved by visiting the node manager’s web UI at `http://node-manager-host:8042/logs/userlogs`.

It is straightforward to write to these logfiles. Anything written to standard output or standard error is directed to the relevant logfile. (Of course, in Streaming, standard output is used for the map or reduce output, so it will not show up in the standard output log.)

In Java, you can write to the task’s *syslog* file if you wish by using the Apache Commons Logging API (or indeed any logging API that can write to log4j). This is shown in [Example 6-13](#).

Example 6-13. An identity mapper that writes to standard output and also uses the Apache Commons Logging API

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.hadoop.mapreduce.Mapper;

public class LoggingIdentityMapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
    extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

    private static final Log LOG = LogFactory.getLog(LoggingIdentityMapper.class);

    @Override
    @SuppressWarnings("unchecked")
    public void map(KEYIN key, VALUEIN value, Context context)
        throws IOException, InterruptedException {
        // Log to stdout file
        System.out.println("Map key: " + key);

        // Log to syslog file
        LOG.info("Map key: " + key);
        if (LOG.isDebugEnabled()) {
            LOG.debug("Map value: " + value);
        }
        context.write((KEYOUT) key, (VALUEOUT) value);
    }
}
```

The default log level is INFO, so DEBUG-level messages do not appear in the *syslog* task logfile. However, sometimes you want to see these messages. To enable this, set `mapreduce.map.log.level` or `mapreduce.reduce.log.level`, as appropriate. For example, in this case, we could set it for the mapper to see the map values in the log as follows:

```
% hadoop jar hadoop-examples.jar LoggingDriver -conf conf/hadoop-cluster.xml \  
-D mapreduce.map.log.level=DEBUG input/ncdc/sample.txt logging-out
```

There are some controls for managing the retention and size of task logs. By default, logs are deleted after a minimum of three hours (you can set this using the `yarn.nodemanager.log.retain-seconds` property, although this is ignored if log aggregation is enabled). You can also set a cap on the maximum size of each logfile using the `mapreduce.task.userlog.limit.kb` property, which is 0 by default, meaning there is no cap.



Sometimes you may need to debug a problem that you suspect is occurring in the JVM running a Hadoop command, rather than on the cluster. You can send DEBUG-level logs to the console by using an invocation like this:

```
% HADOOP_ROOT_LOGGER=DEBUG,console hadoop fs -text /foo/bar
```

Remote Debugging

When a task fails and there is not enough information logged to diagnose the error, you may want to resort to running a debugger for that task. This is hard to arrange when running the job on a cluster, as you don't know which node is going to process which part of the input, so you can't set up your debugger ahead of the failure. However, there are a few other options available:

Reproduce the failure locally

Often the failing task fails consistently on a particular input. You can try to reproduce the problem locally by downloading the file that the task is failing on and running the job locally, possibly using a debugger such as Java's VisualVM.

Use JVM debugging options

A common cause of failure is a Java out of memory error in the task JVM. You can set `mapred.child.java.opts` to include `-XX:-HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/path/to/dumps`. This setting produces a heap dump that can be examined afterward with tools such as *jhat* or the Eclipse Memory Analyzer. Note that the JVM options should be added to the existing memory settings specified by `mapred.child.java.opts`. These are explained in more detail in [“Memory settings in YARN and MapReduce” on page 301](#).

Use task profiling

Java profilers give a lot of insight into the JVM, and Hadoop provides a mechanism to profile a subset of the tasks in a job. See [“Profiling Tasks” on page 175](#).

In some cases, it's useful to keep the intermediate files for a failed task attempt for later inspection, particularly if supplementary dump or profile files are created in the task's working directory. You can set `mapreduce.task.files.preserve.failedtasks` to `true` to keep a failed task's files.

You can keep the intermediate files for successful tasks, too, which may be handy if you want to examine a task that isn't failing. In this case, set the property `mapreduce.task.files.preserve.filepattern` to a regular expression that matches the IDs of the tasks whose files you want to keep.

Another useful property for debugging is `yarn.nodemanager.delete.debug-delay-sec`, which is the number of seconds to wait to delete localized task attempt files, such as the script used to launch the task container JVM. If this is set on the cluster to a reasonably large value (e.g., 600 for 10 minutes), then you have enough time to look at the files before they are deleted.

To examine task attempt files, log into the node that the task failed on and look for the directory for that task attempt. It will be under one of the local MapReduce directories, as set by the `mapreduce.cluster.local.dir` property (covered in more detail in [“Important Hadoop Daemon Properties” on page 296](#)). If this property is a comma-separated

list of directories (to spread load across the physical disks on a machine), you may need to look in all of the directories before you find the directory for that particular task attempt. The task attempt directory is in the following location:

```
mapreduce.cluster.local.dir/usercache/user/appcache/application-ID/output
/task-attempt-ID
```

Tuning a Job

After a job is working, the question many developers ask is, “Can I make it run faster?”

There are a few Hadoop-specific “usual suspects” that are worth checking to see whether they are responsible for a performance problem. You should run through the checklist in [Table 6-3](#) before you start trying to profile or optimize at the task level.

Table 6-3. Tuning checklist

Area	Best practice	Further information
Number of mappers	How long are your mappers running for? If they are only running for a few seconds on average, you should see whether there’s a way to have fewer mappers and make them all run longer—a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using.	“Small files and CombineFileInputFormat” on page 226
Number of reducers	Check that you are using more than a single reducer. Reduce tasks should run for five minutes or so and produce at least a block’s worth of data, as a rule of thumb.	“Choosing the Number of Reducers” on page 217
Combiners	Check whether your job can take advantage of a combiner to reduce the amount of data passing through the shuffle.	“Combiner Functions” on page 34
Intermediate compression	Job execution time can almost always benefit from enabling map output compression.	“Compressing map output” on page 108
Custom serialization	If you are using your own custom <code>Writable</code> objects or custom comparators, make sure you have implemented <code>RawComparator</code> .	“Implementing a RawComparator for speed” on page 123
Shuffle tweaks	The MapReduce shuffle exposes around a dozen tuning parameters for memory management, which may help you wring out the last bit of performance.	“Configuration Tuning” on page 201

Profiling Tasks

Like debugging, profiling a job running on a distributed system such as MapReduce presents some challenges. Hadoop allows you to profile a fraction of the tasks in a job and, as each task completes, pulls down the profile information to your machine for later analysis with standard profiling tools.

Of course, it’s possible, and somewhat easier, to profile a job running in the local job runner. And provided you can run with enough input data to exercise the map and

reduce tasks, this can be a valuable way of improving the performance of your mappers and reducers. There are a couple of caveats, however. The local job runner is a very different environment from a cluster, and the data flow patterns are very different. Optimizing the CPU performance of your code may be pointless if your MapReduce job is I/O-bound (as many jobs are). To be sure that any tuning is effective, you should compare the new execution time with the old one running on a real cluster. Even this is easier said than done, since job execution times can vary due to resource contention with other jobs and the decisions the scheduler makes regarding task placement. To get a good idea of job execution time under these circumstances, perform a series of runs (with and without the change) and check whether any improvement is statistically significant.

It's unfortunately true that some problems (such as excessive memory use) can be reproduced only on the cluster, and in these cases the ability to profile in situ is indispensable.

The HPROF profiler

There are a number of configuration properties to control profiling, which are also exposed via convenience methods on `JobConf`. Enabling profiling is as simple as setting the property `mapreduce.task.profile` to `true`:

```
% hadoop jar hadoop-examples.jar v4.MaxTemperatureDriver \  
  -conf conf/hadoop-cluster.xml \  
  -D mapreduce.task.profile=true \  
  input/ncdc/all max-temp
```

This runs the job as normal, but adds an `-agentlib` parameter to the Java command used to launch the task containers on the node managers. You can control the precise parameter that is added by setting the `mapreduce.task.profile.params` property. The default uses HPROF, a profiling tool that comes with the JDK that, although basic, can give valuable information about a program's CPU and heap usage.

It doesn't usually make sense to profile all tasks in the job, so by default only those with IDs 0, 1, and 2 are profiled (for both maps and reduces). You can change this by setting `mapreduce.task.profile.maps` and `mapreduce.task.profile.reduces` to specify the range of task IDs to profile.

The profile output for each task is saved with the task logs in the *userlogs* subdirectory of the node manager's local log directory (alongside the *syslog*, *stdout*, and *stderr* files), and can be retrieved in the way described in [“Hadoop Logs” on page 172](#), according to whether log aggregation is enabled or not.

MapReduce Workflows

So far in this chapter, you have seen the mechanics of writing a program using MapReduce. We haven't yet considered how to turn a data processing problem into the MapReduce model.

The data processing you have seen so far in this book is to solve a fairly simple problem: finding the maximum recorded temperature for given years. When the processing gets more complex, this complexity is generally manifested by having more MapReduce jobs, rather than having more complex map and reduce functions. In other words, as a rule of thumb, think about adding *more* jobs, rather than adding complexity *to* jobs.

For more complex problems, it is worth considering a higher-level language than MapReduce, such as Pig, Hive, Cascading, Crunch, or Spark. One immediate benefit is that it frees you from having to do the translation into MapReduce jobs, allowing you to concentrate on the analysis you are performing.

Finally, the book *Data-Intensive Text Processing with MapReduce* by Jimmy Lin and Chris Dyer (Morgan & Claypool Publishers, 2010) is a great resource for learning more about MapReduce algorithm design and is highly recommended.

Decomposing a Problem into MapReduce Jobs

Let's look at an example of a more complex problem that we want to translate into a MapReduce workflow.

Imagine that we want to find the mean maximum recorded temperature for every day of the year and every weather station. In concrete terms, to calculate the mean maximum daily temperature recorded by station 029070-99999, say, on January 1, we take the mean of the maximum daily temperatures for this station for January 1, 1901; January 1, 1902; and so on, up to January 1, 2000.

How can we compute this using MapReduce? The computation decomposes most naturally into two stages:

1. *Compute the maximum daily temperature for every station-date pair.*

The MapReduce program in this case is a variant of the maximum temperature program, except that the keys in this case are a composite station-date pair, rather than just the year.

2. *Compute the mean of the maximum daily temperatures for every station-day-month key.*

The mapper takes the output from the previous job (station-date, maximum temperature) records and projects it into (station-day-month, maximum temperature)

records by dropping the year component. The reduce function then takes the mean of the maximum temperatures for each station-day-month key.

The output from the first stage looks like this for the station we are interested in (the *mean_max_daily_temp.sh* script in the examples provides an implementation in Hadoop Streaming):

```
029070-99999 19010101 0
029070-99999 19020101 -94
...
```

The first two fields form the key, and the final column is the maximum temperature from all the readings for the given station and date. The second stage averages these daily maxima over years to yield:

```
029070-99999 0101 -68
```

which is interpreted as saying the mean maximum daily temperature on January 1 for station 029070-99999 over the century is -6.8°C .

It's possible to do this computation in one MapReduce stage, but it takes more work on the part of the programmer.²

The arguments for having more (but simpler) MapReduce stages are that doing so leads to more composable and more maintainable mappers and reducers. Some of the case studies referred to in [Part V](#) cover real-world problems that were solved using MapReduce, and in each case, the data processing task is implemented using two or more MapReduce jobs. The details in that chapter are invaluable for getting a better idea of how to decompose a processing problem into a MapReduce workflow.

It's possible to make map and reduce functions even more composable than we have done. A mapper commonly performs input format parsing, projection (selecting the relevant fields), and filtering (removing records that are not of interest). In the mappers you have seen so far, we have implemented all of these functions in a single mapper. However, there is a case for splitting these into distinct mappers and chaining them into a single mapper using the `ChainMapper` library class that comes with Hadoop. Combined with a `ChainReducer`, you can run a chain of mappers, followed by a reducer and another chain of mappers, in a single MapReduce job.

JobControl

When there is more than one job in a MapReduce workflow, the question arises: how do you manage the jobs so they are executed in order? There are several approaches, and the main consideration is whether you have a linear chain of jobs or a more complex directed acyclic graph (DAG) of jobs.

2. It's an interesting exercise to do this. Hint: use “[Secondary Sort](#)” on page 262.

For a linear chain, the simplest approach is to run each job one after another, waiting until a job completes successfully before running the next:

```
JobClient.runJob(conf1);  
JobClient.runJob(conf2);
```

If a job fails, the `runJob()` method will throw an `IOException`, so later jobs in the pipeline don't get executed. Depending on your application, you might want to catch the exception and clean up any intermediate data that was produced by any previous jobs.

The approach is similar with the new MapReduce API, except you need to examine the Boolean return value of the `waitForCompletion()` method on `Job`: `true` means the job succeeded, and `false` means it failed.

For anything more complex than a linear chain, there are libraries that can help orchestrate your workflow (although they are also suited to linear chains, or even one-off jobs). The simplest is in the `org.apache.hadoop.mapreduce.jobcontrol` package: the `JobControl` class. (There is an equivalent class in the `org.apache.hadoop.mapred.jobcontrol` package, too.) An instance of `JobControl` represents a graph of jobs to be run. You add the job configurations, then tell the `JobControl` instance the dependencies between jobs. You run the `JobControl` in a thread, and it runs the jobs in dependency order. You can poll for progress, and when the jobs have finished, you can query for all the jobs' statuses and the associated errors for any failures. If a job fails, `JobControl` won't run its dependencies.

Apache Oozie

Apache Oozie is a system for running workflows of dependent jobs. It is composed of two main parts: a *workflow engine* that stores and runs workflows composed of different types of Hadoop jobs (MapReduce, Pig, Hive, and so on), and a *coordinator engine* that runs workflow jobs based on predefined schedules and data availability. Oozie has been designed to scale, and it can manage the timely execution of thousands of workflows in a Hadoop cluster, each composed of possibly dozens of constituent jobs.

Oozie makes rerunning failed workflows more tractable, since no time is wasted running successful parts of a workflow. Anyone who has managed a complex batch system knows how difficult it can be to catch up from jobs missed due to downtime or failure, and will appreciate this feature. (Furthermore, coordinator applications representing a single data pipeline may be packaged into a *bundle* and run together as a unit.)

Unlike `JobControl`, which runs on the client machine submitting the jobs, Oozie runs as a service in the cluster, and clients submit workflow definitions for immediate or later execution. In Oozie parlance, a workflow is a DAG of *action nodes* and *control-flow nodes*.

An action node performs a workflow task, such as moving files in HDFS; running a MapReduce, Streaming, Pig, or Hive job; performing a Sqoop import; or running an arbitrary shell script or Java program. A control-flow node governs the workflow execution between actions by allowing such constructs as conditional logic (so different execution branches may be followed depending on the result of an earlier action node) or parallel execution. When the workflow completes, Oozie can make an HTTP callback to the client to inform it of the workflow status. It is also possible to receive callbacks every time the workflow enters or exits an action node.

Defining an Oozie workflow

Workflow definitions are written in XML using the Hadoop Process Definition Language, the specification for which can be found on the [Oozie website](#). **Example 6-14** shows a simple Oozie workflow definition for running a single MapReduce job.

Example 6-14. Oozie workflow definition to run the maximum temperature MapReduce job

```
<workflow-app xmlns="uri:oozie:workflow:0.1" name="max-temp-workflow">
  <start to="max-temp-mr"/>
  <action name="max-temp-mr">
    <map-reduce>
      <job-tracker>${resourceManager}</job-tracker>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="${nameNode}/user/${wf:user()}/output"/>
      </prepare>
      <configuration>
        <property>
          <name>mapred.mapper.new-api</name>
          <value>true</value>
        </property>
        <property>
          <name>mapred.reducer.new-api</name>
          <value>true</value>
        </property>
        <property>
          <name>mapreduce.job.map.class</name>
          <value>MaxTemperatureMapper</value>
        </property>
        <property>
          <name>mapreduce.job.combine.class</name>
          <value>MaxTemperatureReducer</value>
        </property>
        <property>
          <name>mapreduce.job.reduce.class</name>
          <value>MaxTemperatureReducer</value>
        </property>
        <property>
          <name>mapreduce.job.output.key.class</name>

```

```

    <value>org.apache.hadoop.io.Text</value>
  </property>
</property>
  <name>mapreduce.job.output.value.class</name>
  <value>org.apache.hadoop.io.IntWritable</value>
</property>
</property>
  <name>mapreduce.input.fileinputformat.inputdir</name>
  <value>/user/${wf:user()}/input/ncdc/micro</value>
</property>
</property>
  <name>mapreduce.output.fileoutputformat.outputdir</name>
  <value>/user/${wf:user()}/output</value>
</property>
</configuration>
</map-reduce>
<ok to="end" />
<error to="fail" />
</action>
<kill name="fail">
  <message>MapReduce failed, error message[${wf:errorMessage(wf:lastErrorNode())}]
</message>
</kill>
<end name="end" />
</workflow-app>

```

This workflow has three control-flow nodes and one action node: a start control node, a map-reduce action node, a kill control node, and an end control node. The nodes and allowed transitions between them are shown in [Figure 6-4](#).

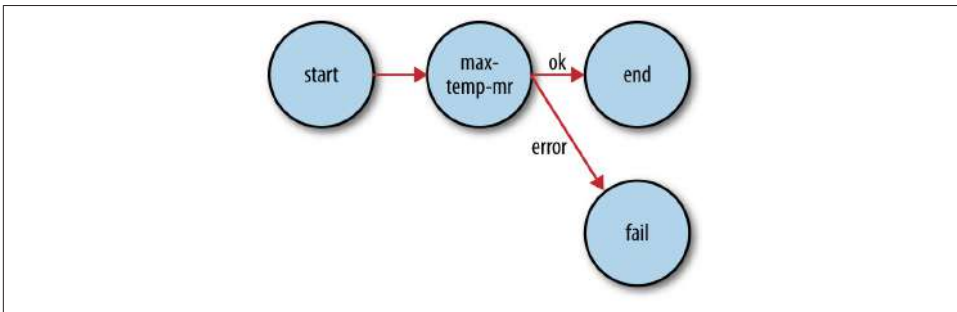


Figure 6-4. Transition diagram of an Oozie workflow

All workflows must have one start and one end node. When the workflow job starts, it transitions to the node specified by the start node (the max-temp-mr action in this example). A workflow job succeeds when it transitions to the end node. However, if the workflow job transitions to a kill node, it is considered to have failed and reports the appropriate error message specified by the message element in the workflow definition.

The bulk of this workflow definition file specifies the `map-reduce` action. The first two elements, `job-tracker` and `name-node`, are used to specify the YARN resource manager (or jobtracker in Hadoop 1) to submit the job to and the namenode (actually a Hadoop filesystem URI) for input and output data. Both are parameterized so that the workflow definition is not tied to a particular cluster (which makes it easy to test). The parameters are specified as workflow job properties at submission time, as we shall see later.



Despite its name, the `job-tracker` element is used to specify a YARN resource manager address and port.

The optional `prepare` element runs before the MapReduce job and is used for directory deletion (and creation, too, if needed, although that is not shown here). By ensuring that the output directory is in a consistent state before running a job, Oozie can safely rerun the action if the job fails.

The MapReduce job to run is specified in the `configuration` element using nested elements for specifying the Hadoop configuration name-value pairs. You can view the MapReduce configuration section as a declarative replacement for the driver classes that we have used elsewhere in this book for running MapReduce programs (such as [Example 2-5](#)).

We have taken advantage of JSP Expression Language (EL) syntax in several places in the workflow definition. Oozie provides a set of functions for interacting with the workflow. For example, `${wf:user()}` returns the name of the user who started the current workflow job, and we use it to specify the correct filesystem path. The Oozie specification lists all the EL functions that Oozie supports.

Packaging and deploying an Oozie workflow application

A workflow application is made up of the workflow definition plus all the associated resources (such as MapReduce JAR files, Pig scripts, and so on) needed to run it. Applications must adhere to a simple directory structure, and are deployed to HDFS so that they can be accessed by Oozie. For this workflow application, we'll put all of the files in a base directory called *max-temp-workflow*, as shown diagrammatically here:

```
max-temp-workflow/  
├── lib/  
│   └── hadoop-examples.jar  
└── workflow.xml
```

The workflow definition file *workflow.xml* must appear in the top level of this directory. JAR files containing the application's MapReduce classes are placed in the *lib* directory.

Workflow applications that conform to this layout can be built with any suitable build tool, such as Ant or Maven; you can find an example in the code that accompanies this book. Once an application has been built, it should be copied to HDFS using regular Hadoop tools. Here is the appropriate command for this application:

```
% hadoop fs -put hadoop-examples/target/max-temp-workflow max-temp-workflow
```

Running an Oozie workflow job

Next, let's see how to run a workflow job for the application we just uploaded. For this we use the *oozie* command-line tool, a client program for communicating with an Oozie server. For convenience, we export the `OOZIE_URL` environment variable to tell the *oozie* command which Oozie server to use (here we're using one running locally):

```
% export OOZIE_URL="http://localhost:11000/oozie"
```

There are lots of subcommands for the *oozie* tool (type `oozie help` to get a list), but we're going to call the `job` subcommand with the `-run` option to run the workflow job:

```
% oozie job -config ch06-mr-dev/src/main/resources/max-temp-workflow.properties \  
-run  
job: 00000001-140911033236814-oozie-oozi-W
```

The `-config` option specifies a local Java properties file containing definitions for the parameters in the workflow XML file (in this case, `nameNode` and `resourceManager`), as well as `oozie.wf.application.path`, which tells Oozie the location of the workflow application in HDFS. Here are the contents of the properties file:

```
nameNode=hdfs://localhost:8020  
resourceManager=localhost:8032  
oozie.wf.application.path=${nameNode}/user/${user.name}/max-temp-workflow
```

To get information about the status of the workflow job, we use the `-info` option, specifying the job ID that was printed by the `run` command earlier (type `oozie job` to get a list of all jobs):

```
% oozie job -info 00000001-140911033236814-oozie-oozi-W
```

The output shows the status: `RUNNING`, `KILLED`, or `SUCCEEDED`. You can also find all this information via Oozie's web UI (<http://localhost:11000/oozie>).

When the job has succeeded, we can inspect the results in the usual way:

```
% hadoop fs -cat output/part-*
1949 111
1950 22
```

This example only scratched the surface of writing Oozie workflows. The documentation on Oozie's website has information about creating more complex workflows, as well as writing and running coordinator jobs.

How MapReduce Works

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

Anatomy of a MapReduce Job Run

You can run a MapReduce job with a single method call: `submit()` on a `Job` object (you can also call `waitForCompletion()`, which submits the job if it hasn't been submitted already, then waits for it to finish).¹ This method call conceals a great deal of processing behind the scenes. This section uncovers the steps Hadoop takes to run a job.

The whole process is illustrated in [Figure 7-1](#). At the highest level, there are five independent entities:²

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

1. In the old MapReduce API, you can call `JobClient.submitJob(conf)` or `JobClient.runJob(conf)`.

2. Not discussed in this section are the job history server daemon (for retaining job history data) and the shuffle handler auxiliary service (for serving map outputs to reduce tasks).

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

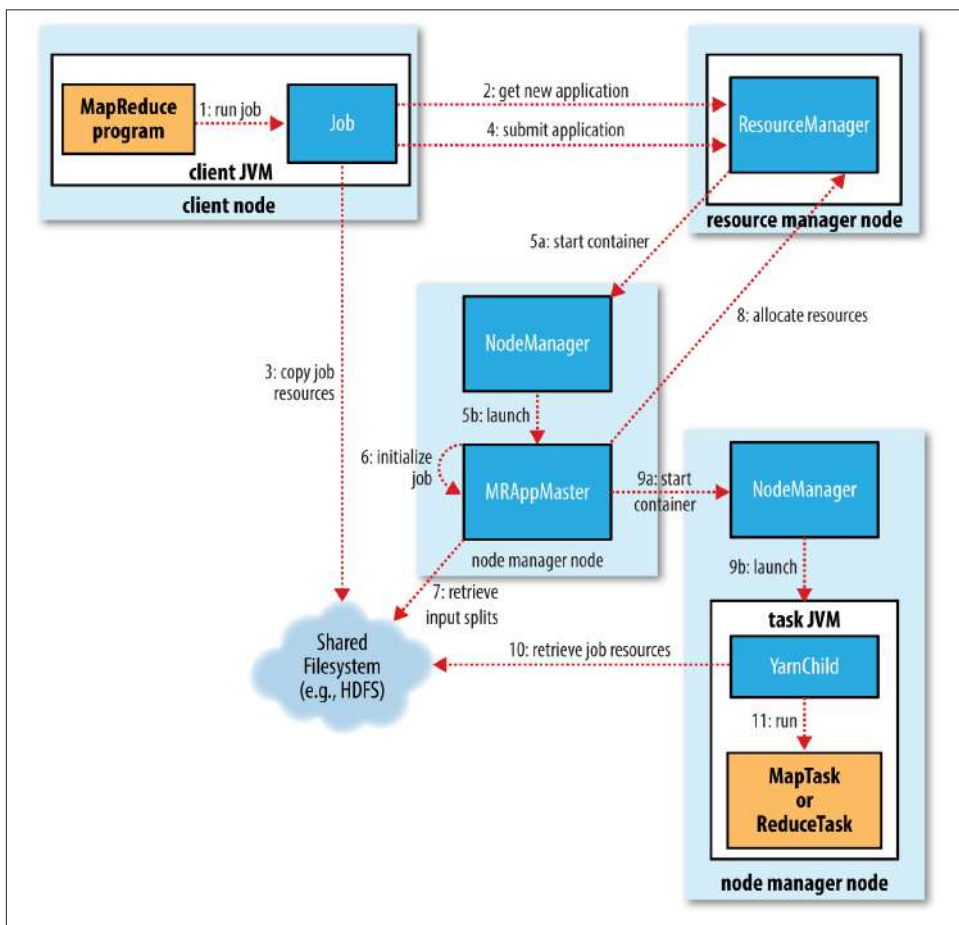


Figure 7-1. How Hadoop runs a MapReduce job

Job Submission

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

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

- Asks the resource manager for a new application ID, used for the MapReduce job ID (step 2).
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program.
- Computes the input splits for the job. If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID (step 3). The job JAR is copied with a high replication factor (controlled by the `mapreduce.client.submit.file.replication` property, which defaults to 10) so that there are lots of copies across the cluster for the node managers to access when they run tasks for the job.
- Submits the job by calling `submitApplication()` on the resource manager (step 4).

Job Initialization

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

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

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

What qualifies as a small job? By default, a small job is one that has less than 10 mappers, only one reducer, and an input size that is less than the size of one HDFS block. (Note that these values may be changed for a job by setting

`mapreduce.job.ubertask.maxmaps`, `mapreduce.job.ubertask.maxreduces`, and `mapreduce.job.ubertask.maxbytes`.) Uber tasks must be enabled explicitly (for an individual job, or across the cluster) by setting `mapreduce.job.ubertask.enable` to `true`.

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

Task Assignment

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

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

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

Task Execution

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

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

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

Streaming

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

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

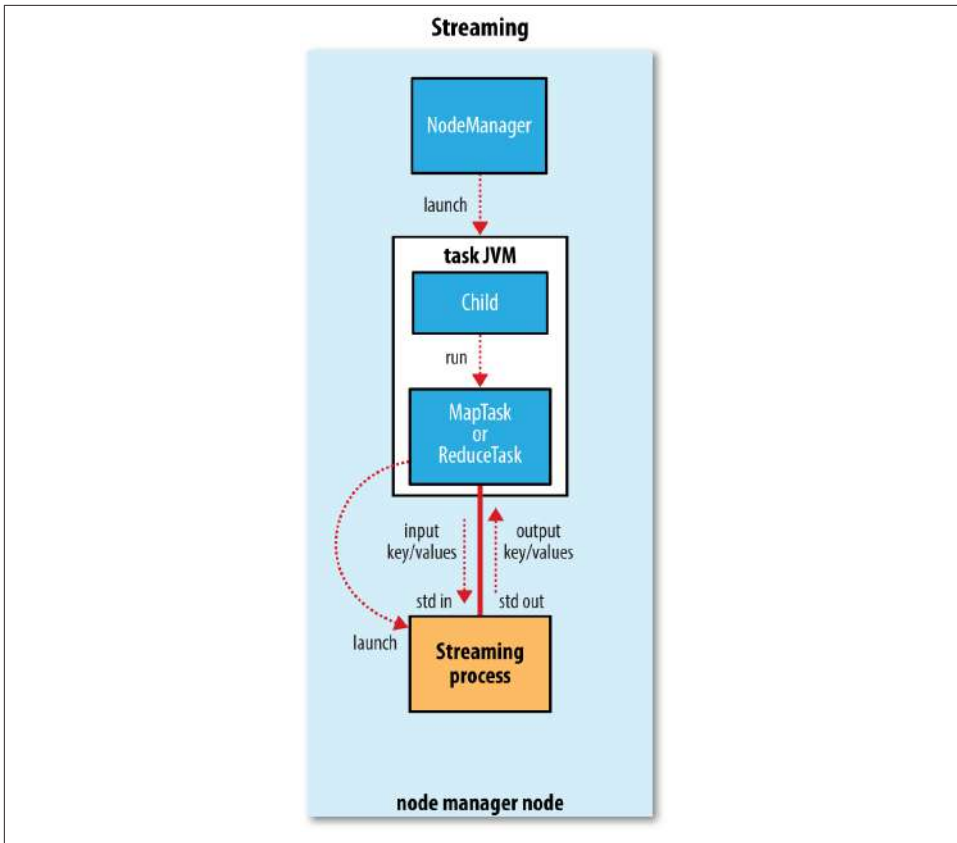


Figure 7-2. The relationship of the Streaming executable to the node manager and the task container

Progress and Status Updates

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

When a task is running, it keeps track of its *progress* (i.e., the proportion of the task completed). For map tasks, this is the proportion of the input that has been processed. For reduce tasks, it's a little more complex, but the system can still estimate the proportion of the reduce input processed. It does this by dividing the total progress into

three parts, corresponding to the three phases of the shuffle (see “[Shuffle and Sort](#)” on [page 197](#)). For example, if the task has run the reducer on half its input, the task’s progress is 5/6, since it has completed the copy and sort phases (1/3 each) and is halfway through the reduce phase (1/6).

What Constitutes Progress in MapReduce?

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

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

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

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

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

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

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

The process is illustrated in [Figure 7-3](#).

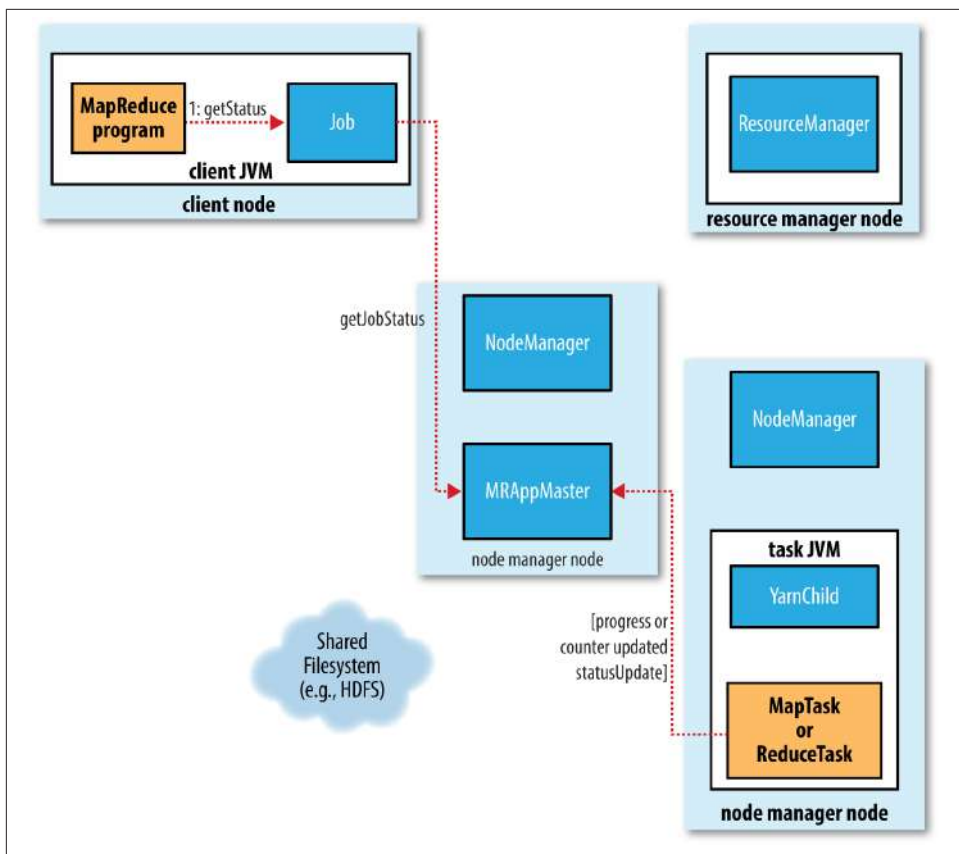


Figure 7-3. How status updates are propagated through the MapReduce system

Job Completion

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

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

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

Failures

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

Task Failure

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

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

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

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

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

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

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

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

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

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

Application Master Failure

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

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

The way recovery works is as follows. An application master sends periodic heartbeats to the resource manager, and in the event of application master failure, the resource manager will detect the failure and start a new instance of the master running in a new container (managed by a node manager). In the case of the MapReduce application

master, it will use the job history to recover the state of the tasks that were already run by the (failed) application so they don't have to be rerun. Recovery is enabled by default, but can be disabled by setting `yarn.app.mapreduce.am.job.recovery.enable` to `false`.

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

Node Manager Failure

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

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

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



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

Resource Manager Failure

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

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

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

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

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

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

Shuffle and Sort

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

The Map Side

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

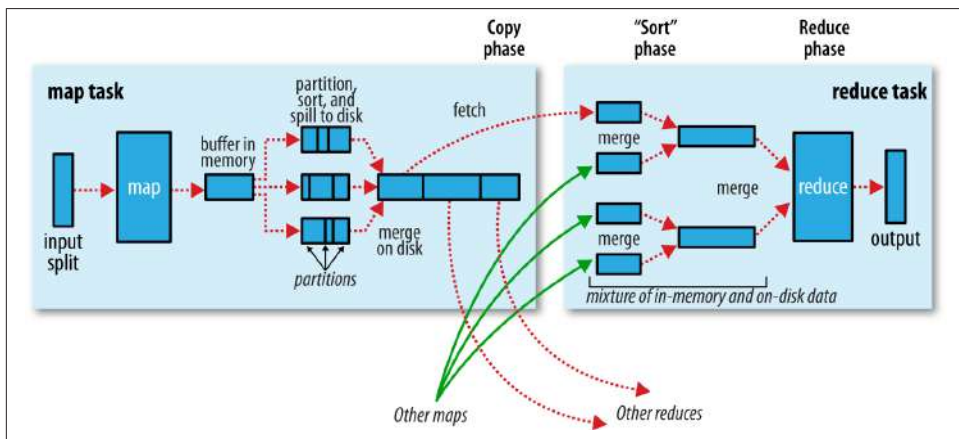


Figure 7-4. Shuffle and sort in MapReduce

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

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

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

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

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

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

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

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

The Reduce Side

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

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



How do reducers know which machines to fetch map output from?

As map tasks complete successfully, they notify their application master using the heartbeat mechanism. Therefore, for a given job, the application master knows the mapping between map outputs and hosts. A thread in the reducer periodically asks the master for map output hosts until it has retrieved them all.

Hosts do not delete map outputs from disk as soon as the first reducer has retrieved them, as the reducer may subsequently fail. Instead, they wait until they are told to delete them by the application master, which is after the job has completed.

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

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

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

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



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

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

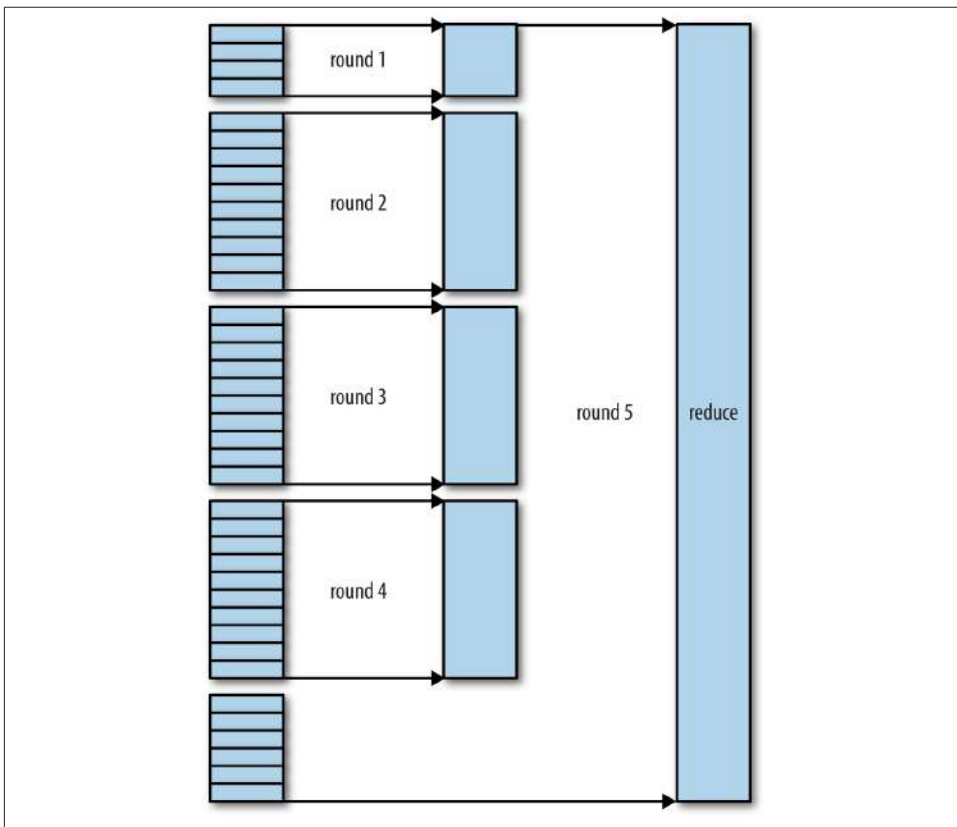


Figure 7-5. Efficiently merging 40 file segments with a merge factor of 10

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

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

Configuration Tuning

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

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

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

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

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

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

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

Table 7-1. Map-side tuning properties

Property name	Type	Default value	Description
<code>mapreduce.task.io.sort.mb</code>	int	100	The size, in megabytes, of the memory buffer to use while sorting map output.
<code>mapreduce.map.sort.spill.percent</code>	float	0.80	The threshold usage proportion for both the map output memory buffer and the record boundaries index to start the process of spilling to disk.
<code>mapreduce.task.io.sort.factor</code>	int	10	The maximum number of streams to merge at once when sorting files. This property is also used in the reduce. It's fairly common to increase this to 100.
<code>mapreduce.map.combine.min spills</code>	int	3	The minimum number of spill files needed for the combiner to run (if a combiner is specified).
<code>mapreduce.map.output.compress</code>	boolean	false	Whether to compress map outputs.
<code>mapreduce.map.output.compress.codec</code>	Class name	<code>org.apache.hadoop.io.compress.DefaultCodec</code>	The compression codec to use for map outputs.
<code>mapreduce.shuffle.max.threads</code>	int	0	The number of worker threads per node manager for serving the map outputs to reducers. This is a cluster-wide setting and cannot be set by individual jobs. 0 means use the Netty default of twice the number of available processors.

Table 7-2. Reduce-side tuning properties

Property name	Type	Default value	Description
<code>mapreduce.reduce.shuffle.parallelcopies</code>	int	5	The number of threads used to copy map outputs to the reducer.
<code>mapreduce.reduce.shuffle.maxfetchfailures</code>	int	10	The number of times a reducer tries to fetch a map output before reporting the error.
<code>mapreduce.task.io.sort.factor</code>	int	10	The maximum number of streams to merge at once when sorting files. This property is also used in the map.
<code>mapreduce.reduce.shuffle.input.buffer.percent</code>	float	0.70	The proportion of total heap size to be allocated to the map outputs buffer during the copy phase of the shuffle.
<code>mapreduce.reduce.shuffle.merge.percent</code>	float	0.66	The threshold usage proportion for the map outputs buffer (defined by <code>mapred.job.shuffle.input.buffer.percent</code>) for starting the process of merging the outputs and spilling to disk.

Property name	Type	Default value	Description
<code>mapreduce.reduce.merge.in mem.threshold</code>	<code>int</code>	1000	The threshold number of map outputs for starting the process of merging the outputs and spilling to disk. A value of 0 or less means there is no threshold, and the spill behavior is governed solely by <code>mapreduce.reduce.shuffle.merge.percent</code> .
<code>mapreduce.reduce.in put.buffer.percent</code>	<code>float</code>	0.0	The proportion of total heap size to be used for retaining map outputs in memory during the reduce. For the reduce phase to begin, the size of map outputs in memory must be no more than this size. By default, all map outputs are merged to disk before the reduce begins, to give the reducers as much memory as possible. However, if your reducers require less memory, this value may be increased to minimize the number of trips to disk.

Task Execution

We saw how the MapReduce system executes tasks in the context of the overall job at the beginning of this chapter, in [“Anatomy of a MapReduce Job Run” on page 185](#). In this section, we’ll look at some more controls that MapReduce users have over task execution.

The Task Execution Environment

Hadoop provides information to a map or reduce task about the environment in which it is running. For example, a map task can discover the name of the file it is processing (see [“File information in the mapper” on page 227](#)), and a map or reduce task can find out the attempt number of the task. The properties in [Table 7-3](#) can be accessed from the job’s configuration, obtained in the old MapReduce API by providing an implementation of the `configure()` method for `Mapper` or `Reducer`, where the configuration is passed in as an argument. In the new API, these properties can be accessed from the context object passed to all methods of the `Mapper` or `Reducer`.

Table 7-3. Task environment properties

Property name	Type	Description	Example
<code>mapreduce.job.id</code>	<code>String</code>	The job ID (see “Job, Task, and Task Attempt IDs” on page 164 for a description of the format)	<code>job_200811201130_0004</code>
<code>mapreduce.task.id</code>	<code>String</code>	The task ID	<code>task_200811201130_0004_m_000003</code>
<code>mapreduce.task.at tempt.id</code>	<code>String</code>	The task attempt ID	<code>attempt_200811201130_0004_m_000003_0</code>

Property name	Type	Description	Example
<code>mapreduce.task.partition</code>	<code>int</code>	The index of the task within the job	3
<code>mapreduce.task.ismap</code>	<code>boolean</code>	Whether this task is a map task	true

Streaming environment variables

Hadoop sets job configuration parameters as environment variables for Streaming programs. However, it replaces nonalphanumeric characters with underscores to make sure they are valid names. The following Python expression illustrates how you can retrieve the value of the `mapreduce.job.id` property from within a Python Streaming script:

```
os.environ["mapreduce_job_id"]
```

You can also set environment variables for the Streaming processes launched by MapReduce by supplying the `-cmdenv` option to the Streaming launcher program (once for each variable you wish to set). For example, the following sets the `MAGIC_PARAMETER` environment variable:

```
-cmdenv MAGIC_PARAMETER=abracadabra
```

Speculative Execution

The MapReduce model is to break jobs into tasks and run the tasks in parallel to make the overall job execution time smaller than it would be if the tasks ran sequentially. This makes the job execution time sensitive to slow-running tasks, as it takes only one slow task to make the whole job take significantly longer than it would have done otherwise. When a job consists of hundreds or thousands of tasks, the possibility of a few straggling tasks is very real.

Tasks may be slow for various reasons, including hardware degradation or software misconfiguration, but the causes may be hard to detect because the tasks still complete successfully, albeit after a longer time than expected. Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another equivalent task as a backup. This is termed *speculative execution* of tasks.

It's important to understand that speculative execution does not work by launching two duplicate tasks at about the same time so they can race each other. This would be wasteful of cluster resources. Rather, the scheduler tracks the progress of all tasks of the same type (map and reduce) in a job, and only launches speculative duplicates for the small proportion that are running significantly slower than the average. When a task completes successfully, any duplicate tasks that are running are killed since they are no longer

needed. So, if the original task completes before the speculative task, the speculative task is killed; on the other hand, if the speculative task finishes first, the original is killed.

Speculative execution is an optimization, and not a feature to make jobs run more reliably. If there are bugs that sometimes cause a task to hang or slow down, relying on speculative execution to avoid these problems is unwise and won't work reliably, since the same bugs are likely to affect the speculative task. You should fix the bug so that the task doesn't hang or slow down.

Speculative execution is turned on by default. It can be enabled or disabled independently for map tasks and reduce tasks, on a cluster-wide basis, or on a per-job basis. The relevant properties are shown in [Table 7-4](#).

Table 7-4. Speculative execution properties

Property name	Type	Default value	Description
<code>mapreduce.map.speculative</code>	boolean	true	Whether extra instances of map tasks may be launched if a task is making slow progress
<code>mapreduce.reduce.speculative</code>	boolean	true	Whether extra instances of reduce tasks may be launched if a task is making slow progress
<code>yarn.app.mapreduce.am.job.speculator.class</code>	Class	<code>org.apache.hadoop.mapreduce.v2.app.speculatelate.DefaultSpeculator</code>	The Speculator class implementing the speculative execution policy (MapReduce 2 only)
<code>yarn.app.mapreduce.am.job.task.estimate.class</code>	Class	<code>org.apache.hadoop.mapreduce.v2.app.speculatelate.LegacyTaskRuntimeEstimator</code>	An implementation of <code>TaskRuntimeEstimator</code> used by Speculator instances that provides estimates for task runtimes (MapReduce 2 only)

Why would you ever want to turn speculative execution off? The goal of speculative execution is to reduce job execution time, but this comes at the cost of cluster efficiency. On a busy cluster, speculative execution can reduce overall throughput, since redundant tasks are being executed in an attempt to bring down the execution time for a single job. For this reason, some cluster administrators prefer to turn it off on the cluster and have users explicitly turn it on for individual jobs. This was especially relevant for older versions of Hadoop, when speculative execution could be overly aggressive in scheduling speculative tasks.

There is a good case for turning off speculative execution for reduce tasks, since any duplicate reduce tasks have to fetch the same map outputs as the original task, and this can significantly increase network traffic on the cluster.

Another reason for turning off speculative execution is for nonidempotent tasks. However, in many cases it is possible to write tasks to be idempotent and use an

OutputCommitter to promote the output to its final location when the task succeeds. This technique is explained in more detail in the next section.

Output Committers

Hadoop MapReduce uses a commit protocol to ensure that jobs and tasks either succeed or fail cleanly. The behavior is implemented by the OutputCommitter in use for the job, which is set in the old MapReduce API by calling the `setOutputCommitter()` on JobConf or by setting `mapred.output.committer.class` in the configuration. In the new MapReduce API, the OutputCommitter is determined by the OutputFormat, via its `getOutputCommitter()` method. The default is FileOutputCommitter, which is appropriate for file-based MapReduce. You can customize an existing OutputCommitter or even write a new implementation if you need to do special setup or cleanup for jobs or tasks.

The OutputCommitter API is as follows (in both the old and new MapReduce APIs):

```
public abstract class OutputCommitter {

    public abstract void setupJob(JobContext jobContext) throws IOException;
    public void commitJob(JobContext jobContext) throws IOException { }
    public void abortJob(JobContext jobContext, JobStatus.State state)
        throws IOException { }

    public abstract void setupTask(TaskAttemptContext taskContext)
        throws IOException;
    public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
        throws IOException;
    public abstract void commitTask(TaskAttemptContext taskContext)
        throws IOException;
    public abstract void abortTask(TaskAttemptContext taskContext)
        throws IOException;

}
```

The `setupJob()` method is called before the job is run, and is typically used to perform initialization. For FileOutputCommitter, the method creates the final output directory, `${mapreduce.output.fileoutputformat.outputdir}`, and a temporary working space for task output, `_temporary`, as a subdirectory underneath it.

If the job succeeds, the `commitJob()` method is called, which in the default file-based implementation deletes the temporary working space and creates a hidden empty marker file in the output directory called `_SUCCESS` to indicate to filesystem clients that the job completed successfully. If the job did not succeed, `abortJob()` is called with a state object indicating whether the job failed or was killed (by a user, for example). In the default implementation, this will delete the job's temporary working space.

The operations are similar at the task level. The `setupTask()` method is called before the task is run, and the default implementation doesn't do anything, because temporary directories named for task outputs are created when the task outputs are written.

The commit phase for tasks is optional and may be disabled by returning `false` from `needsTaskCommit()`. This saves the framework from having to run the distributed commit protocol for the task, and neither `commitTask()` nor `abortTask()` is called. `FileOutputCommitter` will skip the commit phase when no output has been written by a task.

If a task succeeds, `commitTask()` is called, which in the default implementation moves the temporary task output directory (which has the task attempt ID in its name to avoid conflicts between task attempts) to the final output path, `${mapreduce.output.fileoutputformat.outputdir}`. Otherwise, the framework calls `abortTask()`, which deletes the temporary task output directory.

The framework ensures that in the event of multiple task attempts for a particular task, only one will be committed; the others will be aborted. This situation may arise because the first attempt failed for some reason—in which case, it would be aborted, and a later, successful attempt would be committed. It can also occur if two task attempts were running concurrently as speculative duplicates; in this instance, the one that finished first would be committed, and the other would be aborted.

Task side-effect files

The usual way of writing output from map and reduce tasks is by using `OutputCollector` to collect key-value pairs. Some applications need more flexibility than a single key-value pair model, so these applications write output files directly from the map or reduce task to a distributed filesystem, such as HDFS. (There are other ways to produce multiple outputs, too, as described in [“Multiple Outputs” on page 240](#).)

Care needs to be taken to ensure that multiple instances of the same task don't try to write to the same file. As we saw in the previous section, the `OutputCommitter` protocol solves this problem. If applications write side files in their tasks' working directories, the side files for tasks that successfully complete will be promoted to the output directory automatically, whereas failed tasks will have their side files deleted.

A task may find its working directory by retrieving the value of the `mapreduce.task.output.dir` property from the job configuration. Alternatively, a MapReduce program using the Java API may call the `getWorkOutputPath()` static method on `FileOutputFormat` to get the `Path` object representing the working directory. The framework creates the working directory before executing the task, so you don't need to create it.

To take a simple example, imagine a program for converting image files from one format to another. One way to do this is to have a map-only job, where each map is given a set of images to convert (perhaps using `NLineInputFormat`; see [“NLineInputFormat” on](#)

page 234). If a map task writes the converted images into its working directory, they will be promoted to the output directory when the task successfully finishes.

MapReduce Types and Formats

MapReduce has a simple model of data processing: inputs and outputs for the map and reduce functions are key-value pairs. This chapter looks at the MapReduce model in detail, and in particular at how data in various formats, from simple text to structured binary objects, can be used with this model.

MapReduce Types

The map and reduce functions in Hadoop MapReduce have the following general form:

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

In general, the map input key and value types (K1 and V1) are different from the map output types (K2 and V2). However, the reduce input must have the same types as the map output, although the reduce output types may be different again (K3 and V3). The Java API mirrors this general form:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
        // ...
    }
    protected void map(KEYIN key, VALUEIN value,
        Context context) throws IOException, InterruptedException {
        // ...
    }
}

public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
        // ...
    }
    protected void reduce(KEYIN key, Iterable<VALUEIN> values,
        Context context) throws IOException, InterruptedException {
```

```

    // ...
  }
}

```

The context objects are used for emitting key-value pairs, and they are parameterized by the output types so that the signature of the `write()` method is:

```

public void write(KEYOUT key, VALUEOUT value)
    throws IOException, InterruptedException

```

Since `Mapper` and `Reducer` are separate classes, the type parameters have different scopes, and the actual type argument of `KEYIN` (say) in the `Mapper` may be different from the type of the type parameter of the same name (`KEYIN`) in the `Reducer`. For instance, in the maximum temperature example from earlier chapters, `KEYIN` is replaced by `LongWritable` for the `Mapper` and by `Text` for the `Reducer`.

Similarly, even though the map output types and the reduce input types must match, this is not enforced by the Java compiler.

The type parameters are named differently from the abstract types (`KEYIN` versus `K1`, and so on), but the form is the same.

If a combiner function is used, then it has the same form as the reduce function (and is an implementation of `Reducer`), except its output types are the intermediate key and value types (`K2` and `V2`), so they can feed the reduce function:

```

map: (K1, V1) → list(K2, V2)
combiner: (K2, list(V2)) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)

```

Often the combiner and reduce functions are the same, in which case `K3` is the same as `K2`, and `V3` is the same as `V2`.

The partition function operates on the intermediate key and value types (`K2` and `V2`) and returns the partition index. In practice, the partition is determined solely by the key (the value is ignored):

```

partition: (K2, V2) → integer

```

Or in Java:

```

public abstract class Partitioner<KEY, VALUE> {
    public abstract int getPartition(KEY key, VALUE value, int numPartitions);
}

```

MapReduce Signatures in the Old API

In the old API (see [Appendix D](#)), the signatures are very similar and actually name the type parameters K1, V1, and so on, although the constraints on the types are exactly the same in both the old and new APIs:

```
public interface Mapper<K1, V1, K2, V2> extends JobConfigurable, Closeable {
    void map(K1 key, V1 value,
            OutputCollector<K2, V2> output, Reporter reporter) throws IOException;
}

public interface Reducer<K2, V2, K3, V3> extends JobConfigurable, Closeable {
    void reduce(K2 key, Iterator<V2> values,
            OutputCollector<K3, V3> output, Reporter reporter) throws IOException;
}

public interface Partitioner<K2, V2> extends JobConfigurable {
    int getPartition(K2 key, V2 value, int numPartitions);
}
```

So much for the theory. How does this help you configure MapReduce jobs? [Table 8-1](#) summarizes the configuration options for the new API (and [Table 8-2](#) does the same for the old API). It is divided into the properties that determine the types and those that have to be compatible with the configured types.

Input types are set by the input format. So, for instance, a `TextInputFormat` generates keys of type `LongWritable` and values of type `Text`. The other types are set explicitly by calling the methods on the `Job` (or `JobConf` in the old API). If not set explicitly, the intermediate types default to the (final) output types, which default to `LongWritable` and `Text`. So, if `K2` and `K3` are the same, you don't need to call `setMapOutputKeyClass()`, because it falls back to the type set by calling `setOutputKeyClass()`. Similarly, if `V2` and `V3` are the same, you only need to use `setOutputValueClass()`.

It may seem strange that these methods for setting the intermediate and final output types exist at all. After all, why can't the types be determined from a combination of the mapper and the reducer? The answer has to do with a limitation in Java generics: type erasure means that the type information isn't always present at runtime, so Hadoop has to be given it explicitly. This also means that it's possible to configure a MapReduce job with incompatible types, because the configuration isn't checked at compile time. The settings that have to be compatible with the MapReduce types are listed in the lower part of [Table 8-1](#). Type conflicts are detected at runtime during job execution, and for this reason, it is wise to run a test job using a small amount of data to flush out and fix any type incompatibilities.

Table 8-1. Configuration of MapReduce types in the new API

Property	Job setter method	Input types			Intermediate types			Output types		
		K1	V1	K2	V2	K3	V3			
Properties for configuring types:										
mapreduce.job.inputformat.class	setInputFormatClass()	•	•							
mapreduce.map.output.key.class	setMapOutputKeyClass()			•						
mapreduce.map.output.value.class	setMapOutputValueClass()				•					
mapreduce.job.output.key.class	setOutputKeyClass()					•				
mapreduce.job.output.value.class	setOutputValueClass()						•			
Properties that must be consistent with the types:										
mapreduce.job.map.class	setMapperClass()	•	•	•	•					
mapreduce.job.combine.class	setCombinerClass()			•		•				
mapreduce.job.partitioner.class	setPartitionerClass()			•		•				
mapreduce.job.output.key.comparator.class	setSortComparatorClass()				•					
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()				•					
mapreduce.job.reduce.class	setReducerClass()			•		•	•			
mapreduce.job.outputformat.class	setOutputFormatClass()							•		

Table 8-2. Configuration of MapReduce types in the old API

Property	JobConf setter method	Input types			Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3	
Properties for configuring types:								
mapred.input.format.class	setInputFormat()	.	.					
mapred.mapoutput.key.class	setMapOutputKeyClass()			.				
mapred.mapoutput.value.class	setMapOutputValueClass()				.			
mapred.output.key.class	setOutputKeyClass()					.		
mapred.output.value.class	setOutputValueClass()						.	
Properties that must be consistent with the types:								
mapred.mapper.class	setMapperClass()			
mapred.map.runner.class	setMapRunnerClass()			
mapred.combiner.class	setCombinerClass()			.	.			
mapred.partitioner.class	setPartitionerClass()			.	.			
mapred.output.key.comparator.class	setOutputKeyComparatorClass()			.				
mapred.output.value.groupfn.class	setOutputValueGroupingComparator()			.				
mapred.reducer.class	setReducerClass()			
mapred.output.format.class	setOutputFormat()					.	.	

The Default MapReduce Job

What happens when you run MapReduce without setting a mapper or a reducer? Let's try it by running this minimal MapReduce program:

```
public class MinimalMapReduce extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.printf("Usage: %s [generic options] <input> <output>\n",
                getClass().getSimpleName());
            ToolRunner.printGenericCommandUsage(System.err);
            return -1;
        }

        Job job = new Job(getConf());
        job.setJarByClass(getClass());
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        return job.waitForCompletion(true) ? 0 : 1;
    }

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

The only configuration that we set is an input path and an output path. We run it over a subset of our weather data with the following:

```
% hadoop MinimalMapReduce "input/ncdc/all/190{1,2}.gz" output
```

We do get some output: one file named *part-r-00000* in the output directory. Here's what the first few lines look like (truncated to fit the page):

```
0→0029029070999991901010106004+64333+023450FM-12+000599999V0202701N01591...
0→0035029070999991902010106004+64333+023450FM-12+000599999V0201401N01181...
135→0029029070999991901010113004+64333+023450FM-12+000599999V0202901N00821...
141→0035029070999991902010113004+64333+023450FM-12+000599999V0201401N01181...
270→0029029070999991901010120004+64333+023450FM-12+000599999V0209991C00001...
282→0035029070999991902010120004+64333+023450FM-12+000599999V0201401N01391...
```

Each line is an integer followed by a tab character, followed by the original weather data record. Admittedly, it's not a very useful program, but understanding how it produces its output does provide some insight into the defaults that Hadoop uses when running MapReduce jobs. [Example 8-1](#) shows a program that has exactly the same effect as `MinimalMapReduce`, but explicitly sets the job settings to their defaults.

Example 8-1. A minimal MapReduce driver, with the defaults explicitly set

```
public class MinimalMapReduceWithDefaults extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
        if (job == null) {
            return -1;
        }

        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(Mapper.class);

        job.setMapOutputKeyClass(LongWritable.class);
        job.setMapOutputValueClass(Text.class);

        job.setPartitionerClass(HashPartitioner.class);

        job.setNumReduceTasks(1);
        job.setReducerClass(Reducer.class);

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

        job.setOutputFormatClass(TextOutputFormat.class);

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

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

We've simplified the first few lines of the `run()` method by extracting the logic for printing usage and setting the input and output paths into a helper method. Almost all MapReduce drivers take these two arguments (input and output), so reducing the boilerplate code here is a good thing. Here are the relevant methods in the `JobBuilder` class for reference:

```
public static Job parseInputAndOutput(Tool tool, Configuration conf,
    String[] args) throws IOException {

    if (args.length != 2) {
        printUsage(tool, "<input> <output>");
        return null;
    }
    Job job = new Job(conf);
    job.setJarByClass(tool.getClass());
```

```

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

    public static void printUsage(Tool tool, String extraArgsUsage) {
        System.err.printf("Usage: %s [genericOptions] %s\n\n",
            tool.getClass().getSimpleName(), extraArgsUsage);
        GenericOptionsParser.printGenericCommandUsage(System.err);
    }

```

Going back to `MinimalMapReduceWithDefaults` in [Example 8-1](#), although there are many other default job settings, the ones **bolded** are those most central to running a job. Let's go through them in turn.

The default input format is `TextInputFormat`, which produces keys of type `LongWritable` (the offset of the beginning of the line in the file) and values of type `Text` (the line of text). This explains where the integers in the final output come from: they are the line offsets.

The default mapper is just the `Mapper` class, which writes the input key and value unchanged to the output:

```

public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

    protected void map(KEYIN key, VALUEIN value,
        Context context) throws IOException, InterruptedException {
        context.write((KEYOUT) key, (VALUEOUT) value);
    }
}

```

`Mapper` is a generic type, which allows it to work with any key or value types. In this case, the map input and output key is of type `LongWritable`, and the map input and output value is of type `Text`.

The default partitioner is `HashPartitioner`, which hashes a record's key to determine which partition the record belongs in. Each partition is processed by a reduce task, so the number of partitions is equal to the number of reduce tasks for the job:

```

public class HashPartitioner<K, V> extends Partitioner<K, V> {

    public int getPartition(K key, V value,
        int numReduceTasks) {
        return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
    }
}

```

The key's hash code is turned into a nonnegative integer by bitwise ANDing it with the largest integer value. It is then reduced modulo the number of partitions to find the index of the partition that the record belongs in.

By default, there is a single reducer, and therefore a single partition; the action of the partitioner is irrelevant in this case since everything goes into one partition. However, it is important to understand the behavior of `HashPartitioner` when you have more than one reduce task. Assuming the key's hash function is a good one, the records will be allocated evenly across reduce tasks, with all records that share the same key being processed by the same reduce task.

You may have noticed that we didn't set the number of map tasks. The reason for this is that the number is equal to the number of splits that the input is turned into, which is driven by the size of the input and the file's block size (if the file is in HDFS). The options for controlling split size are discussed in [“FileInputFormat input splits” on page 224](#).

Choosing the Number of Reducers

The single reducer default is something of a gotcha for new users to Hadoop. Almost all real-world jobs should set this to a larger number; otherwise, the job will be very slow since all the intermediate data flows through a single reduce task.

Choosing the number of reducers for a job is more of an art than a science. Increasing the number of reducers makes the reduce phase shorter, since you get more parallelism. However, if you take this too far, you can have lots of small files, which is suboptimal. One rule of thumb is to aim for reducers that each run for five minutes or so, and which produce at least one HDFS block's worth of output.

The default reducer is `Reducer`, again a generic type, which simply writes all its input to its output:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

    protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context
        Context context) throws IOException, InterruptedException {
        for (VALUEIN value: values) {
            context.write((KEYOUT) key, (VALUEOUT) value);
        }
    }
}
```

For this job, the output key is `LongWritable` and the output value is `Text`. In fact, all the keys for this MapReduce program are `LongWritable` and all the values are `Text`, since these are the input keys and values, and the map and reduce functions are both identity functions, which by definition preserve type. Most MapReduce programs, however, don't use the same key or value types throughout, so you need to configure the job to declare the types you are using, as described in the previous section.

Records are sorted by the MapReduce system before being presented to the reducer. In this case, the keys are sorted numerically, which has the effect of interleaving the lines from the input files into one combined output file.

The default output format is `TextOutputFormat`, which writes out records, one per line, by converting keys and values to strings and separating them with a tab character. This is why the output is tab-separated: it is a feature of `TextOutputFormat`.

The default Streaming job

In Streaming, the default job is similar, but not identical, to the Java equivalent. The basic form is:

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \  
-input input/ncdc/sample.txt \  
-output output \  
-mapper /bin/cat
```

When we specify a non-Java mapper and the default text mode is in effect (`-io text`), Streaming does something special. It doesn't pass the key to the mapper process; it just passes the value. (For other input formats, the same effect can be achieved by setting `stream.map.input.ignoreKey` to true.) This is actually very useful because the key is just the line offset in the file and the value is the line, which is all most applications are interested in. The overall effect of this job is to perform a sort of the input.

With more of the defaults spelled out, the command looks like this (notice that Streaming uses the old MapReduce API classes):

```
% hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \  
-input input/ncdc/sample.txt \  
-output output \  
-inputformat org.apache.hadoop.mapred.TextInputFormat \  
-mapper /bin/cat \  
-partitioner org.apache.hadoop.mapred.lib.HashPartitioner \  
-numReduceTasks 1 \  
-reducer org.apache.hadoop.mapred.lib.IdentityReducer \  
-outputformat org.apache.hadoop.mapred.TextOutputFormat  
-io text
```

The `-mapper` and `-reducer` arguments take a command or a Java class. A combiner may optionally be specified using the `-combiner` argument.

Keys and values in Streaming

A Streaming application can control the separator that is used when a key-value pair is turned into a series of bytes and sent to the map or reduce process over standard input. The default is a tab character, but it is useful to be able to change it in the case that the keys or values themselves contain tab characters.

Similarly, when the map or reduce writes out key-value pairs, they may be separated by a configurable separator. Furthermore, the key from the output can be composed of more than the first field: it can be made up of the first *n* fields (defined by `stream.num.map.output.key.fields` or `stream.num.reduce.output.key.fields`), with the value being the remaining fields. For example, if the output from a Streaming process was `a,b,c` (with a comma as the separator), and *n* was 2, the key would be parsed as `a,b` and the value as `c`.

Separators may be configured independently for maps and reduces. The properties are listed in [Table 8-3](#) and shown in a diagram of the data flow path in [Figure 8-1](#).

These settings do not have any bearing on the input and output formats. For example, if `stream.reduce.output.field.separator` were set to be a colon, say, and the reduce stream process wrote the line `a:b` to standard out, the Streaming reducer would know to extract the key as `a` and the value as `b`. With the standard `TextOutputFormat`, this record would be written to the output file with a tab separating `a` and `b`. You can change the separator that `TextOutputFormat` uses by setting `mapreduce.output.textoutputformat.separator`.

Table 8-3. Streaming separator properties

Property name	Type	Default value	Description
<code>stream.map.input.field.separator</code>	String	<code>\t</code>	The separator to use when passing the input key and value strings to the stream map process as a stream of bytes
<code>stream.map.output.field.separator</code>	String	<code>\t</code>	The separator to use when splitting the output from the stream map process into key and value strings for the map output
<code>stream.num.map.output.key.fields</code>	int	1	The number of fields separated by <code>stream.map.output.field.separator</code> to treat as the map output key
<code>stream.reduce.input.field.separator</code>	String	<code>\t</code>	The separator to use when passing the input key and value strings to the stream reduce process as a stream of bytes
<code>stream.reduce.output.field.separator</code>	String	<code>\t</code>	The separator to use when splitting the output from the stream reduce process into key and value strings for the final reduce output
<code>stream.num.reduce.output.key.fields</code>	int	1	The number of fields separated by <code>stream.reduce.output.field.separator</code> to treat as the reduce output key

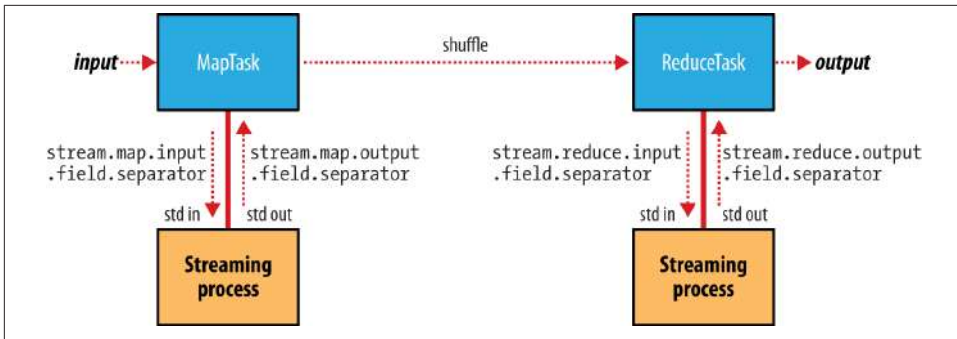


Figure 8-1. Where separators are used in a Streaming MapReduce job

Input Formats

Hadoop can process many different types of data formats, from flat text files to databases. In this section, we explore the different formats available.

Input Splits and Records

As we saw in [Chapter 2](#), an input split is a chunk of the input that is processed by a single map. Each map processes a single split. Each split is divided into records, and the map processes each record—a key-value pair—in turn. Splits and records are logical: there is nothing that requires them to be tied to files, for example, although in their most common incarnations, they are. In a database context, a split might correspond to a range of rows from a table and a record to a row in that range (this is precisely the case with `DBInputFormat`, which is an input format for reading data from a relational database).

Input splits are represented by the Java class `InputSplit` (which, like all of the classes mentioned in this section, is in the `org.apache.hadoop.mapreduce` package):¹

```
public abstract class InputSplit {
    public abstract long getLength() throws IOException, InterruptedException;
    public abstract String[] getLocations() throws IOException,
        InterruptedException;
}
```

An `InputSplit` has a length in bytes and a set of storage locations, which are just host-name strings. Notice that a split doesn't contain the input data; it is just a reference to the data. The storage locations are used by the MapReduce system to place map tasks as close to the split's data as possible, and the size is used to order the splits so that the

1. But see the classes in `org.apache.hadoop.mapred` for the old MapReduce API counterparts.

largest get processed first, in an attempt to minimize the job runtime (this is an instance of a greedy approximation algorithm).

As a MapReduce application writer, you don't need to deal with `InputSplits` directly, as they are created by an `InputFormat` (an `InputFormat` is responsible for creating the input splits and dividing them into records). Before we see some concrete examples of `InputFormats`, let's briefly examine how it is used in MapReduce. Here's the interface:

```
public abstract class InputFormat<K, V> {
    public abstract List<InputSplit> getSplits(JobContext context)
        throws IOException, InterruptedException;

    public abstract RecordReader<K, V>
        createRecordReader(InputSplit split, TaskAttemptContext context)
            throws IOException, InterruptedException;
}
```

The client running the job calculates the splits for the job by calling `getSplits()`, then sends them to the application master, which uses their storage locations to schedule map tasks that will process them on the cluster. The map task passes the split to the `createRecordReader()` method on `InputFormat` to obtain a `RecordReader` for that split. A `RecordReader` is little more than an iterator over records, and the map task uses one to generate record key-value pairs, which it passes to the map function. We can see this by looking at the Mapper's `run()` method:

```
public void run(Context context) throws IOException, InterruptedException {
    setup(context);
    while (context.nextKeyValue()) {
        map(context.getCurrentKey(), context.getCurrentValue(), context);
    }
    cleanup(context);
}
```

After running `setup()`, the `nextKeyValue()` is called repeatedly on the `Context` (which delegates to the identically named method on the `RecordReader`) to populate the key and value objects for the mapper. The key and value are retrieved from the `RecordReader` by way of the `Context` and are passed to the `map()` method for it to do its work. When the reader gets to the end of the stream, the `nextKeyValue()` method returns `false`, and the map task runs its `cleanup()` method and then completes.



Although it's not shown in the code snippet, for reasons of efficiency, `RecordReader` implementations will return the *same* key and value objects on each call to `getCurrentKey()` and `getCurrentValue()`. Only the contents of these objects are changed by the reader's `nextKeyValue()` method. This can be a surprise to users, who might expect keys and values to be immutable and not to be reused. This causes problems when a reference to a key or value object is retained outside the `map()` method, as its value can change without warning. If you need to do this, make a copy of the object you want to hold on to. For example, for a `Text` object, you can use its copy constructor: `new Text(value)`.

The situation is similar with reducers. In this case, the value objects in the reducer's iterator are reused, so you need to copy any that you need to retain between calls to the iterator (see [Example 9-11](#)).

Finally, note that the `Mapper`'s `run()` method is public and may be customized by users. `MultiThreadedMapper` is an implementation that runs mappers concurrently in a configurable number of threads (set by `mapreduce.mapper.multithreadedmapper.threads`). For most data processing tasks, it confers no advantage over the default implementation. However, for mappers that spend a long time processing each record—because they contact external servers, for example—it allows multiple mappers to run in one JVM with little contention.

FileInputFormat

`FileInputFormat` is the base class for all implementations of `InputFormat` that use files as their data source (see [Figure 8-2](#)). It provides two things: a place to define which files are included as the input to a job, and an implementation for generating splits for the input files. The job of dividing splits into records is performed by subclasses.

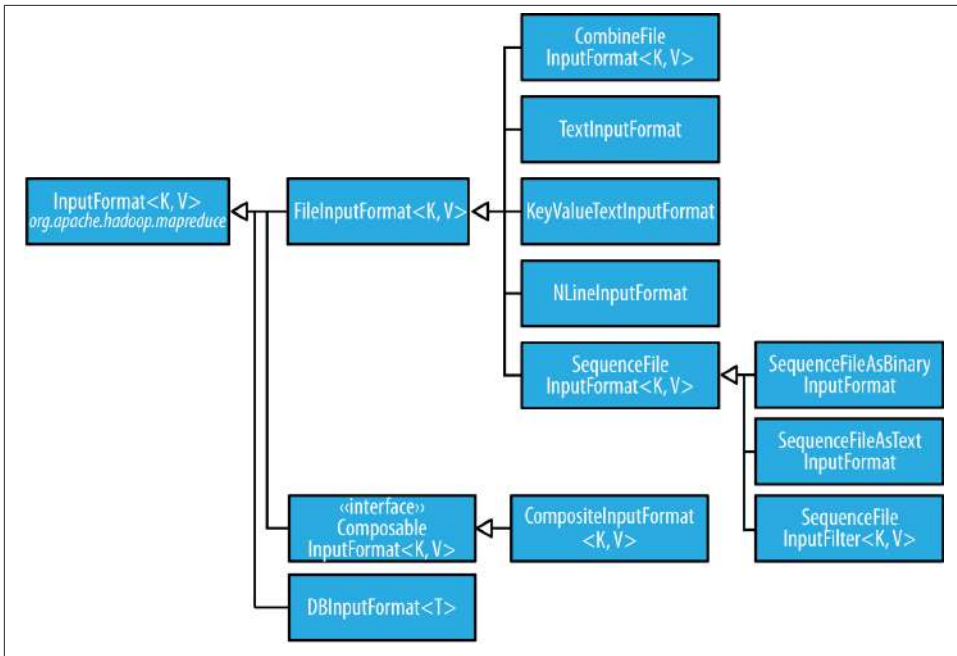


Figure 8-2. *InputFormat* class hierarchy

FileInputFormat input paths

The input to a job is specified as a collection of paths, which offers great flexibility in constraining the input. *FileInputFormat* offers four static convenience methods for setting a Job's input paths:

```

public static void addInputPath(Job job, Path path)
public static void addInputPaths(Job job, String commaSeparatedPaths)
public static void setInputPaths(Job job, Path... inputPaths)
public static void setInputPaths(Job job, String commaSeparatedPaths)

```

The `addInputPath()` and `addInputPaths()` methods add a path or paths to the list of inputs. You can call these methods repeatedly to build the list of paths. The `setInputPaths()` methods set the entire list of paths in one go (replacing any paths set on the Job in previous calls).

A path may represent a file, a directory, or, by using a glob, a collection of files and directories. A path representing a directory includes all the files in the directory as input to the job. See [“File patterns” on page 66](#) for more on using globs.



The contents of a directory specified as an input path are not processed recursively. In fact, the directory should only contain files. If the directory contains a subdirectory, it will be interpreted as a file, which will cause an error. The way to handle this case is to use a file glob or a filter to select only the files in the directory based on a name pattern. Alternatively, `mapreduce.input.fileinputformat.input.dir.recursive` can be set to `true` to force the input directory to be read recursively.

The `add` and `set` methods allow files to be specified by inclusion only. To exclude certain files from the input, you can set a filter using the `setInputPathFilter()` method on `FileInputFormat`. Filters are discussed in more detail in “[PathFilter](#)” on page 67.

Even if you don’t set a filter, `FileInputFormat` uses a default filter that excludes hidden files (those whose names begin with a dot or an underscore). If you set a filter by calling `setInputPathFilter()`, it acts in addition to the default filter. In other words, only nonhidden files that are accepted by your filter get through.

Paths and filters can be set through configuration properties, too ([Table 8-4](#)), which can be handy for Streaming jobs. Setting paths is done with the `-input` option for the Streaming interface, so setting paths directly usually is not needed.

Table 8-4. Input path and filter properties

Property name	Type	Default value	Description
<code>mapreduce.input.fileinputformat.input.dir</code>	Comma-separated paths	None	The input files for a job. Paths that contain commas should have those commas escaped by a backslash character. For example, the glob <code>{a,b}</code> would be escaped as <code>{a\,b}</code> .
<code>mapreduce.input.pathfilter.class</code>	<code>PathFilter</code> classname	None	The filter to apply to the input files for a job.

FileInputFormat input splits

Given a set of files, how does `FileInputFormat` turn them into splits? `FileInputFormat` splits only large files—here, “large” means larger than an HDFS block. The split size is normally the size of an HDFS block, which is appropriate for most applications; however, it is possible to control this value by setting various Hadoop properties, as shown in [Table 8-5](#).

Table 8-5. Properties for controlling split size

Property name	Type	Default value	Description
<code>mapreduce.input.fileinputformat.split.min.size</code>	<code>int</code>	1	The smallest valid size in bytes for a file split
<code>mapreduce.input.fileinputformat.split.max.size</code> ^a	<code>long</code>	<code>Long.MAX_VALUE</code> (i.e., 9223372036854775807)	The largest valid size in bytes for a file split
<code>dfs.blocksize</code>	<code>long</code>	128 MB (i.e., 134217728)	The size of a block in HDFS in bytes

^a This property is not present in the old MapReduce API (with the exception of `CombineFileInputFormat`). Instead, it is calculated indirectly as the size of the total input for the job, divided by the guide number of map tasks specified by `mapreduce.job.maps` (or the `setNumMapTasks()` method on `JobConf`). Because the number of map tasks defaults to 1, this makes the maximum split size the size of the input.

The minimum split size is usually 1 byte, although some formats have a lower bound on the split size. (For example, sequence files insert sync entries every so often in the stream, so the minimum split size has to be large enough to ensure that every split has a sync point to allow the reader to resynchronize with a record boundary. See “[Reading a SequenceFile](#)” on page 129.)

Applications may impose a minimum split size. By setting this to a value larger than the block size, they can force splits to be larger than a block. There is no good reason for doing this when using HDFS, because doing so will increase the number of blocks that are not local to a map task.

The maximum split size defaults to the maximum value that can be represented by a Java long type. It has an effect only when it is less than the block size, forcing splits to be smaller than a block.

The split size is calculated by the following formula (see the `computeSplitSize()` method in `FileInputFormat`):

```
max(minimumSize, min(maximumSize, blockSize))
```

and by default:

```
minimumSize < blockSize < maximumSize
```

so the split size is `blockSize`. Various settings for these parameters and how they affect the final split size are illustrated in [Table 8-6](#).

Table 8-6. Examples of how to control the split size

Minimum split size	Maximum split size	Block size	Split size	Comment
1 (default)	Long.MAX_VALUE (default)	128 MB (default)	128 MB	By default, the split size is the same as the default block size.
1 (default)	Long.MAX_VALUE (default)	256 MB	256 MB	The most natural way to increase the split size is to have larger blocks in HDFS, either by setting <code>dfs.blocksize</code> or by configuring this on a per-file basis at file construction time.
256 MB	Long.MAX_VALUE (default)	128 MB (default)	256 MB	Making the minimum split size greater than the block size increases the split size, but at the cost of locality.
1 (default)	64 MB	128 MB (default)	64 MB	Making the maximum split size less than the block size decreases the split size.

Small files and CombineFileInputFormat

Hadoop works better with a small number of large files than a large number of small files. One reason for this is that `FileInputFormat` generates splits in such a way that each split is all or part of a single file. If the file is very small (“small” means significantly smaller than an HDFS block) and there are a lot of them, each map task will process very little input, and there will be a lot of them (one per file), each of which imposes extra bookkeeping overhead. Compare a 1 GB file broken into eight 128 MB blocks with 10,000 or so 100 KB files. The 10,000 files use one map each, and the job time can be tens or hundreds of times slower than the equivalent one with a single input file and eight map tasks.

The situation is alleviated somewhat by `CombineFileInputFormat`, which was designed to work well with small files. Where `FileInputFormat` creates a split per file, `CombineFileInputFormat` packs many files into each split so that each mapper has more to process. Crucially, `CombineFileInputFormat` takes node and rack locality into account when deciding which blocks to place in the same split, so it does not compromise the speed at which it can process the input in a typical MapReduce job.

Of course, if possible, it is still a good idea to avoid the many small files case, because MapReduce works best when it can operate at the transfer rate of the disks in the cluster, and processing many small files increases the number of seeks that are needed to run a job. Also, storing large numbers of small files in HDFS is wasteful of the namenode’s memory. One technique for avoiding the many small files case is to merge small files into larger files by using a sequence file, as in [Example 8-4](#); with this approach, the keys can act as filenames (or a constant such as `NullWritable`, if not needed) and the values as file contents. But if you already have a large number of small files in HDFS, then `CombineFileInputFormat` is worth trying.



CombineFileInputFormat isn't just good for small files. It can bring benefits when processing large files, too, since it will generate one split per node, which may be made up of multiple blocks. Essentially, CombineFileInputFormat decouples the amount of data that a mapper consumes from the block size of the files in HDFS.

Preventing splitting

Some applications don't want files to be split, as this allows a single mapper to process each input file in its entirety. For example, a simple way to check if all the records in a file are sorted is to go through the records in order, checking whether each record is not less than the preceding one. Implemented as a map task, this algorithm will work only if one map processes the whole file.²

There are a couple of ways to ensure that an existing file is not split. The first (quick-and-dirty) way is to increase the minimum split size to be larger than the largest file in your system. Setting it to its maximum value, `Long.MAX_VALUE`, has this effect. The second is to subclass the concrete subclass of `FileInputFormat` that you want to use, to override the `isSplittable()` method³ to return `false`. For example, here's a nonsplittable `TextInputFormat`:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class NonSplittableTextInputFormat extends TextInputFormat {
    @Override
    protected boolean isSplittable(JobContext context, Path file) {
        return false;
    }
}
```

File information in the mapper

A mapper processing a file input split can find information about the split by calling the `getInputSplit()` method on the Mapper's Context object. When the input format derives from `FileInputFormat`, the `InputSplit` returned by this method can be cast to a `FileSplit` to access the file information listed in [Table 8-7](#).

In the old MapReduce API, and the Streaming interface, the same file split information is made available through properties that can be read from the mapper's configuration.

2. This is how the mapper in `SortValidator.RecordStatsChecker` is implemented.
3. In the method name `isSplittable()`, "splittable" has a single "t." It is usually spelled "splittable," which is the spelling I have used in this book.

(In the old MapReduce API this is achieved by implementing `configure()` in your Mapper implementation to get access to the `JobConf` object.)

In addition to the properties in [Table 8-7](#), all mappers and reducers have access to the properties listed in “[The Task Execution Environment](#)” on page 203.

Table 8-7. File split properties

FileSplit method	Property name	Type	Description
<code>getPath()</code>	<code>mapreduce.map.input.file</code>	Path/ String	The path of the input file being processed
<code>getStart()</code>	<code>mapreduce.map.input.start</code>	long	The byte offset of the start of the split from the beginning of the file
<code>getLength()</code>	<code>mapreduce.map.input.length</code>	long	The length of the split in bytes

In the next section, we’ll see how to use a `FileSplit` when we need to access the split’s filename.

Processing a whole file as a record

A related requirement that sometimes crops up is for mappers to have access to the full contents of a file. Not splitting the file gets you part of the way there, but you also need to have a `RecordReader` that delivers the file contents as the value of the record. The listing for `WholeFileInputFormat` in [Example 8-2](#) shows a way of doing this.

Example 8-2. An `InputFormat` for reading a whole file as a record

```
public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

    @Override
    protected boolean isSplittable(JobContext context, Path file) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(
        InputSplit split, TaskAttemptContext context) throws IOException,
        InterruptedException {
        WholeFileRecordReader reader = new WholeFileRecordReader();
        reader.initialize(split, context);
        return reader;
    }
}
```

`WholeFileInputFormat` defines a format where the keys are not used, represented by `NullWritable`, and the values are the file contents, represented by `BytesWritable` instances. It defines two methods. First, the format is careful to specify that input files should never be split, by overriding `isSplittable()` to return `false`. Second, we

implement `createRecordReader()` to return a custom implementation of `RecordReader`, which appears in [Example 8-3](#).

Example 8-3. The `RecordReader` used by `WholeFileInputFormat` for reading a whole file as a record

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context)
        throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLength()];
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);
            FSDataInputStream in = null;
            try {
                in = fs.open(file);
                IOUtils.readFully(in, contents, 0, contents.length);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
            processed = true;
            return true;
        }
        return false;
    }

    @Override
    public NullWritable getCurrentKey() throws IOException, InterruptedException {
        return NullWritable.get();
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException,
        InterruptedException {
        return value;
    }

    @Override
```

```

public float getProgress() throws IOException {
    return processed ? 1.0f : 0.0f;
}

@Override
public void close() throws IOException {
    // do nothing
}
}

```

WholeFileRecordReader is responsible for taking a FileSplit and converting it into a single record, with a null key and a value containing the bytes of the file. Because there is only a single record, WholeFileRecordReader has either processed it or not, so it maintains a Boolean called processed. If the file has not been processed when the nextKeyValue() method is called, then we open the file, create a byte array whose length is the length of the file, and use the Hadoop IOUtils class to slurp the file into the byte array. Then we set the array on the BytesWritable instance that was passed into the next() method, and return true to signal that a record has been read.

The other methods are straightforward bookkeeping methods for accessing the current key and value types and getting the progress of the reader, and a close() method, which is invoked by the MapReduce framework when the reader is done.

To demonstrate how WholeFileInputFormat can be used, consider a MapReduce job for packaging small files into sequence files, where the key is the original filename and the value is the content of the file. The listing is in [Example 8-4](#).

Example 8-4. A MapReduce program for packaging a collection of small files as a single SequenceFile

```

public class SmallFilesToSequenceFileConverter extends Configured
    implements Tool {

    static class SequenceFileMapper
        extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {

        private Text filenameKey;

        @Override
        protected void setup(Context context) throws IOException,
            InterruptedException {
            InputSplit split = context.getInputSplit();
            Path path = ((FileSplit) split).getPath();
            filenameKey = new Text(path.toString());
        }

        @Override
        protected void map(NullWritable key, BytesWritable value, Context context)
            throws IOException, InterruptedException {
            context.write(filenameKey, value);
        }
    }
}

```



```

    }
}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setInputFormatClass(WholeFileInputFormat.class);
    job.setOutputFormatClass(SequenceFileOutputFormat.class);

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

    job.setMapperClass(SequenceFileMapper.class);

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

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

```

Because the input format is a `WholeFileInputFormat`, the mapper only has to find the filename for the input file split. It does this by casting the `InputSplit` from the context to a `FileSplit`, which has a method to retrieve the file path. The path is stored in a `Text` object for the key. The reducer is the identity (not explicitly set), and the output format is a `SequenceFileOutputFormat`.

Here's a run on a few small files. We've chosen to use two reducers, so we get two output sequence files:

```
% hadoop jar hadoop-examples.jar SmallFilesToSequenceFileConverter \
-conf conf/hadoop-localhost.xml -D mapreduce.job.reduces=2 \
input/smallfiles output
```

Two part files are created, each of which is a sequence file. We can inspect these with the `-text` option to the filesystem shell:

```
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00000
hdfs://localhost/user/tom/input/smallfiles/a 61 61 61 61 61 61 61 61 61 61
hdfs://localhost/user/tom/input/smallfiles/c 63 63 63 63 63 63 63 63 63 63
hdfs://localhost/user/tom/input/smallfiles/e
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00001
hdfs://localhost/user/tom/input/smallfiles/b 62 62 62 62 62 62 62 62 62 62
hdfs://localhost/user/tom/input/smallfiles/d 64 64 64 64 64 64 64 64 64 64
hdfs://localhost/user/tom/input/smallfiles/f 66 66 66 66 66 66 66 66 66 66
```

The input files were named *a*, *b*, *c*, *d*, *e*, and *f*, and each contained 10 characters of the corresponding letter (so, for example, *a* contained 10 “a” characters), except *e*, which was empty. We can see this in the textual rendering of the sequence files, which prints the filename followed by the hex representation of the file.



There’s at least one way we could improve this program. As mentioned earlier, having one mapper per file is inefficient, so subclassing `CombineFileInputFormat` instead of `FileInputFormat` would be a better approach.

Text Input

Hadoop excels at processing unstructured text. In this section, we discuss the different `InputFormats` that Hadoop provides to process text.

`TextInputFormat`

`TextInputFormat` is the default `InputFormat`. Each record is a line of input. The key, a `LongWritable`, is the byte offset within the file of the beginning of the line. The value is the contents of the line, excluding any line terminators (e.g., newline or carriage return), and is packaged as a `Text` object. So, a file containing the following text:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

is divided into one split of four records. The records are interpreted as the following key-value pairs:

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

Clearly, the keys are *not* line numbers. This would be impossible to implement in general, in that a file is broken into splits at byte, not line, boundaries. Splits are processed independently. Line numbers are really a sequential notion. You have to keep a count of lines as you consume them, so knowing the line number within a split would be possible, but not within the file.

However, the offset within the file of each line is known by each split independently of the other splits, since each split knows the size of the preceding splits and just adds this onto the offsets within the split to produce a global file offset. The offset is usually sufficient for applications that need a unique identifier for each line. Combined with the file’s name, it is unique within the filesystem. Of course, if all the lines are a fixed width, calculating the line number is simply a matter of dividing the offset by the width.

The Relationship Between Input Splits and HDFS Blocks

The logical records that `FileInputFormats` define usually do not fit neatly into HDFS blocks. For example, a `TextInputFormat`'s logical records are lines, which will cross HDFS boundaries more often than not. This has no bearing on the functioning of your program—lines are not missed or broken, for example—but it's worth knowing about because it does mean that data-local maps (that is, maps that are running on the same host as their input data) will perform some remote reads. The slight overhead this causes is not normally significant.

Figure 8-3 shows an example. A single file is broken into lines, and the line boundaries do not correspond with the HDFS block boundaries. Splits honor logical record boundaries (in this case, lines), so we see that the first split contains line 5, even though it spans the first and second block. The second split starts at line 6.

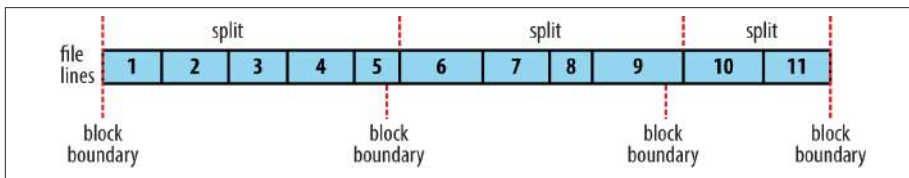


Figure 8-3. Logical records and HDFS blocks for `TextInputFormat`

Controlling the maximum line length. If you are using one of the text input formats discussed here, you can set a maximum expected line length to safeguard against corrupted files. Corruption in a file can manifest itself as a very long line, which can cause out-of-memory errors and then task failure. By setting `mapreduce.input.linerecordreader.line.maxlength` to a value in bytes that fits in memory (and is comfortably greater than the length of lines in your input data), you ensure that the record reader will skip the (long) corrupt lines without the task failing.

`KeyValueTextInputFormat`

`TextInputFormat`'s keys, being simply the offsets within the file, are not normally very useful. It is common for each line in a file to be a key-value pair, separated by a delimiter such as a tab character. For example, this is the kind of output produced by `TextOutputFormat`, Hadoop's default `OutputFormat`. To interpret such files correctly, `KeyValueTextInputFormat` is appropriate.

You can specify the separator via the `mapreduce.input.keyvaluelinerecordreader.key.value.separator` property. It is a tab character by default. Consider the following input file, where `→` represents a (horizontal) tab character:

```
line1→On the top of the Crumpetty Tree
line2→The Quangle Wangle sat,
line3→But his face you could not see,
line4→On account of his Beaver Hat.
```

Like in the `TextInputFormat` case, the input is in a single split comprising four records, although this time the keys are the Text sequences before the tab in each line:

```
(line1, On the top of the Crumpetty Tree)
(line2, The Quangle Wangle sat,)
(line3, But his face you could not see,)
(line4, On account of his Beaver Hat.)
```

NLineInputFormat

With `TextInputFormat` and `KeyValueTextInputFormat`, each mapper receives a variable number of lines of input. The number depends on the size of the split and the length of the lines. If you want your mappers to receive a fixed number of lines of input, then `NLineInputFormat` is the `InputFormat` to use. Like with `TextInputFormat`, the keys are the byte offsets within the file and the values are the lines themselves.

N refers to the number of lines of input that each mapper receives. With N set to 1 (the default), each mapper receives exactly one line of input. The `mapreduce.input.lineinputformat.linespermap` property controls the value of N . By way of example, consider these four lines again:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

If, for example, N is 2, then each split contains two lines. One mapper will receive the first two key-value pairs:

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
```

And another mapper will receive the second two key-value pairs:

```
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

The keys and values are the same as those that `TextInputFormat` produces. The difference is in the way the splits are constructed.

Usually, having a map task for a small number of lines of input is inefficient (due to the overhead in task setup), but there are applications that take a small amount of input data and run an extensive (i.e., CPU-intensive) computation for it, then emit their output. Simulations are a good example. By creating an input file that specifies input parameters, one per line, you can perform a *parameter sweep*: run a set of simulations in parallel to find how a model varies as the parameter changes.



If you have long-running simulations, you may fall afoul of task timeouts. When a task doesn't report progress for more than 10 minutes, the application master assumes it has failed and aborts the process (see [“Task Failure” on page 193](#)).

The best way to guard against this is to report progress periodically, by writing a status message or incrementing a counter, for example. See [“What Constitutes Progress in MapReduce?” on page 191](#).

Another example is using Hadoop to bootstrap data loading from multiple data sources, such as databases. You create a “seed” input file that lists the data sources, one per line. Then each mapper is allocated a single data source, and it loads the data from that source into HDFS. The job doesn't need the reduce phase, so the number of reducers should be set to zero (by calling `setNumReduceTasks()` on `Job`). Furthermore, MapReduce jobs can be run to process the data loaded into HDFS. See [Appendix C](#) for an example.

XML

Most XML parsers operate on whole XML documents, so if a large XML document is made up of multiple input splits, it is a challenge to parse these individually. Of course, you can process the entire XML document in one mapper (if it is not too large) using the technique in [“Processing a whole file as a record” on page 228](#).

Large XML documents that are composed of a series of “records” (XML document fragments) can be broken into these records using simple string or regular-expression matching to find the start and end tags of records. This alleviates the problem when the document is split by the framework because the next start tag of a record is easy to find by simply scanning from the start of the split, just like `TextInputFormat` finds newline boundaries.

Hadoop comes with a class for this purpose called `StreamXmlRecordReader` (which is in the `org.apache.hadoop.streaming.mapreduce` package, although it can be used outside of Streaming). You can use it by setting your input format to `StreamInputFormat` and setting the `stream.recordreader.class` property to `org.apache.hadoop.streaming.mapreduce.StreamXmlRecordReader`. The reader is configured by setting job configuration properties to tell it the patterns for the start and end tags (see the class documentation for details).⁴

To take an example, Wikipedia provides dumps of its content in XML form, which are appropriate for processing in parallel with MapReduce using this approach. The data is contained in one large XML wrapper document, which contains a series of elements,

4. See [Mahout's `XmlInputFormat`](#) for an improved XML input format.

such as page elements that contain a page's content and associated metadata. Using `StreamXMLRecordReader`, the page elements can be interpreted as records for processing by a mapper.

Binary Input

Hadoop MapReduce is not restricted to processing textual data. It has support for binary formats, too.

SequenceFileInputFormat

Hadoop's sequence file format stores sequences of binary key-value pairs. Sequence files are well suited as a format for MapReduce data because they are splittable (they have sync points so that readers can synchronize with record boundaries from an arbitrary point in the file, such as the start of a split), they support compression as a part of the format, and they can store arbitrary types using a variety of serialization frameworks. (These topics are covered in “[SequenceFile](#)” on page 127.)

To use data from sequence files as the input to MapReduce, you can use `SequenceFileInputFormat`. The keys and values are determined by the sequence file, and you need to make sure that your map input types correspond. For example, if your sequence file has `IntWritable` keys and `Text` values, like the one created in [Chapter 5](#), then the map signature would be `Mapper<IntWritable, Text, K, V>`, where `K` and `V` are the types of the map's output keys and values.



Although its name doesn't give it away, `SequenceFileInputFormat` can read map files as well as sequence files. If it finds a directory where it was expecting a sequence file, `SequenceFileInputFormat` assumes that it is reading a map file and uses its datafile. This is why there is no `MapFileInputFormat` class.

SequenceFileAsTextInputFormat

`SequenceFileAsTextInputFormat` is a variant of `SequenceFileInputFormat` that converts the sequence file's keys and values to `Text` objects. The conversion is performed by calling `toString()` on the keys and values. This format makes sequence files suitable input for Streaming.

SequenceFileAsBinaryInputFormat

`SequenceFileAsBinaryInputFormat` is a variant of `SequenceFileInputFormat` that retrieves the sequence file's keys and values as opaque binary objects. They are encapsulated as `BytesWritable` objects, and the application is free to interpret the underlying byte array as it pleases. In combination with a process that creates sequence files with `SequenceFile.Writer's` `appendRaw()` method or

`SequenceFileAsBinaryOutputFormat`, this provides a way to use any binary data types with MapReduce (packaged as a sequence file), although plugging into Hadoop's serialization mechanism is normally a cleaner alternative (see “[Serialization Frameworks](#)” on page 126).

FixedLengthInputFormat

`FixedLengthInputFormat` is for reading fixed-width binary records from a file, when the records are not separated by delimiters. The record size must be set via `fixedlengthinputformat.record.length`.

Multiple Inputs

Although the input to a MapReduce job may consist of multiple input files (constructed by a combination of file globs, filters, and plain paths), all of the input is interpreted by a single `InputFormat` and a single `Mapper`. What often happens, however, is that the data format evolves over time, so you have to write your mapper to cope with all of your legacy formats. Or you may have data sources that provide the same type of data but in different formats. This arises in the case of performing joins of different datasets; see “[Reduce-Side Joins](#)” on page 270. For instance, one might be tab-separated plain text, and the other a binary sequence file. Even if they are in the same format, they may have different representations, and therefore need to be parsed differently.

These cases are handled elegantly by using the `MultipleInputs` class, which allows you to specify which `InputFormat` and `Mapper` to use on a per-path basis. For example, if we had weather data from the UK Met Office⁵ that we wanted to combine with the NCDC data for our maximum temperature analysis, we might set up the input as follows:

```
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

This code replaces the usual calls to `FileInputFormat.addInputPath()` and `job.setMapperClass()`. Both the Met Office and NCDC data are text based, so we use `TextInputFormat` for each. But the line format of the two data sources is different, so we use two different mappers. The `MaxTemperatureMapper` reads NCDC input data and extracts the year and temperature fields. The `MetOfficeMaxTemperatureMapper` reads Met Office input data and extracts the year and temperature fields. The important thing is that the map outputs have the same types, since the reducers (which are all of the same type) see the aggregated map outputs and are not aware of the different mappers used to produce them.

5. Met Office data is generally available only to the research and academic community. However, there is a small amount of monthly weather station data available at <http://www.metoffice.gov.uk/climate/uk/stationdata/>.

The `MultipleInputs` class has an overloaded version of `addInputPath()` that doesn't take a mapper:

```
public static void addInputPath(Job job, Path path,  
                                Class<? extends InputFormat> inputFormatClass)
```

This is useful when you only have one mapper (set using the Job's `setMapperClass()` method) but multiple input formats.

Database Input (and Output)

`DBInputFormat` is an input format for reading data from a relational database, using JDBC. Because it doesn't have any sharding capabilities, you need to be careful not to overwhelm the database from which you are reading by running too many mappers. For this reason, it is best used for loading relatively small datasets, perhaps for joining with larger datasets from HDFS using `MultipleInputs`. The corresponding output format is `DBOutputFormat`, which is useful for dumping job outputs (of modest size) into a database.

For an alternative way of moving data between relational databases and HDFS, consider using Sqoop, which is described in [Chapter 15](#).

HBase's `TableInputFormat` is designed to allow a MapReduce program to operate on data stored in an HBase table. `TableOutputFormat` is for writing MapReduce outputs into an HBase table.

Output Formats

Hadoop has output data formats that correspond to the input formats covered in the previous section. The `OutputFormat` class hierarchy appears in [Figure 8-4](#).

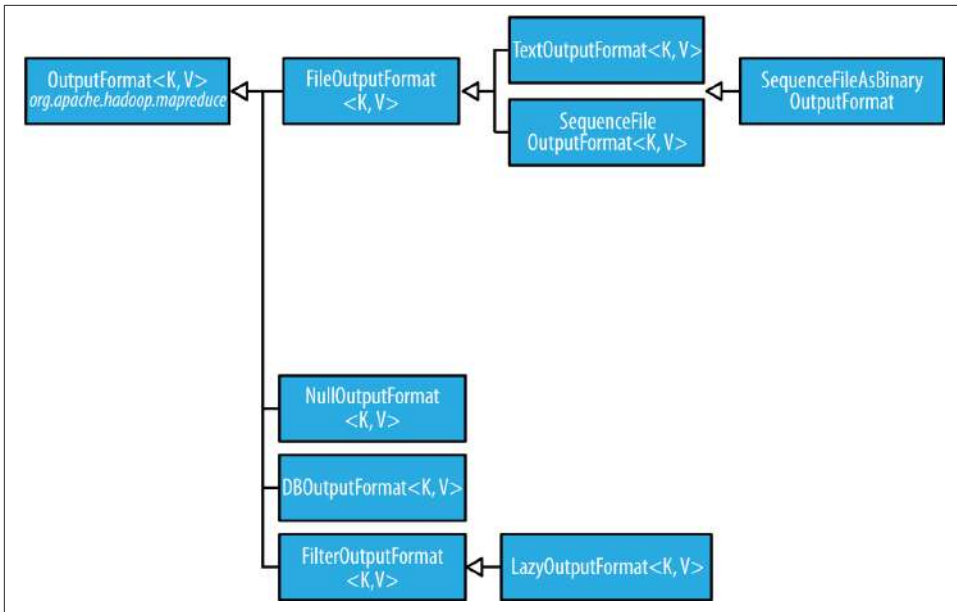


Figure 8-4. *OutputFormat* class hierarchy

Text Output

The default output format, `TextOutputFormat`, writes records as lines of text. Its keys and values may be of any type, since `TextOutputFormat` turns them to strings by calling `toString()` on them. Each key-value pair is separated by a tab character, although that may be changed using the `mapreduce.output.textoutputformat.separator` property. The counterpart to `TextOutputFormat` for reading in this case is `KeyValueTextInputFormat`, since it breaks lines into key-value pairs based on a configurable separator (see “[KeyValueTextInputFormat](#)” on page 233).

You can suppress the key or the value from the output (or both, making this output format equivalent to `NullOutputFormat`, which emits nothing) using a `NullWritable` type. This also causes no separator to be written, which makes the output suitable for reading in using `TextInputFormat`.

Binary Output

SequenceFileOutputFormat

As the name indicates, `SequenceFileOutputFormat` writes sequence files for its output. This is a good choice of output if it forms the input to a further MapReduce job, since it is compact and is readily compressed. Compression is controlled via the static methods on `SequenceFileOutputFormat`, as described in “[Using Compression in MapReduce](#)”

on page 107. For an example of how to use `SequenceFileOutputFormat`, see “Sorting” on page 255.

SequenceFileAsBinaryOutputFormat

`SequenceFileAsBinaryOutputFormat`—the counterpart to `SequenceFileAsBinaryInputFormat`—writes keys and values in raw binary format into a sequence file container.

MapFileOutputFormat

`MapFileOutputFormat` writes map files as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.



The reduce *input* keys are guaranteed to be sorted, but the output keys are under the control of the reduce function, and there is nothing in the general MapReduce contract that states that the reduce *output* keys have to be ordered in any way. The extra constraint of sorted reduce output keys is just needed for `MapFileOutputFormat`.

Multiple Outputs

`FileOutputFormat` and its subclasses generate a set of files in the output directory. There is one file per reducer, and files are named by the partition number: *part-r-00000*, *part-r-00001*, and so on. Sometimes there is a need to have more control over the naming of the files or to produce multiple files per reducer. MapReduce comes with the `MultipleOutputs` class to help you do this.⁶

An example: Partitioning data

Consider the problem of partitioning the weather dataset by weather station. We would like to run a job whose output is one file per station, with each file containing all the records for that station.

One way of doing this is to have a reducer for each weather station. To arrange this, we need to do two things. First, write a partitioner that puts records from the same weather station into the same partition. Second, set the number of reducers on the job to be the number of weather stations. The partitioner would look like this:

6. The old MapReduce API includes two classes for producing multiple outputs: `MultipleOutputFormat` and `MultipleOutputs`. In a nutshell, `MultipleOutputs` is more fully featured, but `MultipleOutputFormat` has more control over the output directory structure and file naming. `MultipleOutputs` in the new API combines the best features of the two multiple output classes in the old API. The code on this book's website includes old API equivalents of the examples in this section using both `MultipleOutputs` and `MultipleOutputFormat`.

```

public class StationPartitioner extends Partitioner<LongWritable, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    public int getPartition(LongWritable key, Text value, int numPartitions) {
        parser.parse(value);
        return getPartition(parser.getStationId());
    }

    private int getPartition(String stationId) {
        ...
    }
}

```

The `getPartition(String)` method, whose implementation is not shown, turns the station ID into a partition index. To do this, it needs a list of all the station IDs; it then just returns the index of the station ID in the list.

There are two drawbacks to this approach. The first is that since the number of partitions needs to be known before the job is run, so does the number of weather stations. Although the NCDC provides metadata about its stations, there is no guarantee that the IDs encountered in the data will match those in the metadata. A station that appears in the metadata but not in the data wastes a reduce task. Worse, a station that appears in the data but not in the metadata doesn't get a reduce task; it has to be thrown away. One way of mitigating this problem would be to write a job to extract the unique station IDs, but it's a shame that we need an extra job to do this.

The second drawback is more subtle. It is generally a bad idea to allow the number of partitions to be rigidly fixed by the application, since this can lead to small or uneven-sized partitions. Having many reducers doing a small amount of work isn't an efficient way of organizing a job; it's much better to get reducers to do more work and have fewer of them, as the overhead in running a task is then reduced. Uneven-sized partitions can be difficult to avoid, too. Different weather stations will have gathered a widely varying amount of data; for example, compare a station that opened one year ago to one that has been gathering data for a century. If a few reduce tasks take significantly longer than the others, they will dominate the job execution time and cause it to be longer than it needs to be.



There are two special cases when it does make sense to allow the application to set the number of partitions (or equivalently, the number of reducers):

Zero reducers

This is a vacuous case: there are no partitions, as the application needs to run only map tasks.

One reducer

It can be convenient to run small jobs to combine the output of previous jobs into a single file. This should be attempted only when the amount of data is small enough to be processed comfortably by one reducer.

It is much better to let the cluster drive the number of partitions for a job, the idea being that the more cluster resources there are available, the faster the job can complete. This is why the default `HashPartitioner` works so well: it works with any number of partitions and ensures each partition has a good mix of keys, leading to more evenly sized partitions.

If we go back to using `HashPartitioner`, each partition will contain multiple stations, so to create a file per station, we need to arrange for each reducer to write multiple files. This is where `MultipleOutputs` comes in.

MultipleOutputs

`MultipleOutputs` allows you to write data to files whose names are derived from the output keys and values, or in fact from an arbitrary string. This allows each reducer (or mapper in a map-only job) to create more than a single file. Filenames are of the form *name-m-nnnnn* for map outputs and *name-r-nnnnn* for reduce outputs, where *name* is an arbitrary name that is set by the program and *nnnnn* is an integer designating the part number, starting from 00000. The part number ensures that outputs written from different partitions (mappers or reducers) do not collide in the case of the same name.

The program in [Example 8-5](#) shows how to use `MultipleOutputs` to partition the dataset by station.

Example 8-5. Partitioning whole dataset into files named by the station ID using `MultipleOutputs`

```
public class PartitionByStationUsingMultipleOutputs extends Configured
    implements Tool {

    static class StationMapper
        extends Mapper<LongWritable, Text, Text, Text> {

        private NcdcRecordParser parser = new NcdcRecordParser();
```

```

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

static class MultipleOutputsReducer
    extends Reducer<Text, Text, NullWritable, Text> {

    private MultipleOutputs<NullWritable, Text> multipleOutputs;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {
        multipleOutputs = new MultipleOutputs<NullWritable, Text>(context);
    }

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        for (Text value : values) {
            multipleOutputs.write(NullWritable.get(), value, key.toString());
        }
    }

    @Override
    protected void cleanup(Context context)
        throws IOException, InterruptedException {
        multipleOutputs.close();
    }
}

@Override
public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
        return -1;
    }

    job.setMapperClass(StationMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setReducerClass(MultipleOutputsReducer.class);
    job.setOutputKeyClass(NullWritable.class);

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

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

```

```
}
}
```

In the reducer, which is where we generate the output, we construct an instance of `MultipleOutputs` in the `setup()` method and assign it to an instance variable. We then use the `MultipleOutputs` instance in the `reduce()` method to write to the output, in place of the context. The `write()` method takes the key and value, as well as a name. We use the station identifier for the name, so the overall effect is to produce output files with the naming scheme *station_identifier-r-nnnnn*.

In one run, the first few output files were named as follows:

```
output/010010-99999-r-00027
output/010050-99999-r-00013
output/010100-99999-r-00015
output/010280-99999-r-00014
output/010550-99999-r-00000
output/010980-99999-r-00011
output/011060-99999-r-00025
output/012030-99999-r-00029
output/012350-99999-r-00018
output/012620-99999-r-00004
```

The base path specified in the `write()` method of `MultipleOutputs` is interpreted relative to the output directory, and because it may contain file path separator characters (`/`), it's possible to create subdirectories of arbitrary depth. For example, the following modification partitions the data by station and year so that each year's data is contained in a directory named by the station ID (such as *029070-99999/1901/part-r-00000*):

```
@Override
protected void reduce(Text key, Iterable<Text> values, Context context)
    throws IOException, InterruptedException {
    for (Text value : values) {
        parser.parse(value);
        String basePath = String.format("%s/%s/part",
            parser.getStationId(), parser.getYear());
        multipleOutputs.write(NullWritable.get(), value, basePath);
    }
}
```

`MultipleOutputs` delegates to the mapper's `OutputFormat`. In this example it's a `TextOutputFormat`, but more complex setups are possible. For example, you can create named outputs, each with its own `OutputFormat` and key and value types (which may differ from the output types of the mapper or reducer). Furthermore, the mapper or reducer (or both) may write to multiple output files for each record processed. Consult the Java documentation for more information.

Lazy Output

`FileOutputFormat` subclasses will create output (*part-r-nnnnn*) files, even if they are empty. Some applications prefer that empty files not be created, which is where `LazyOutputFormat` helps. It is a wrapper output format that ensures that the output file is created only when the first record is emitted for a given partition. To use it, call its `setOutputFormatClass()` method with the `JobConf` and the underlying output format.

Streaming supports a `-lazyOutput` option to enable `LazyOutputFormat`.

Database Output

The output formats for writing to relational databases and to HBase are mentioned in “[Database Input \(and Output\)](#)” on page 238.