CHAPTER 6
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 (note that you can also call `waitForCompletion()`, which will submit the job if it hasn't been submitted already, then wait 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.

We saw in Chapter 5 that the way Hadoop executes a MapReduce program depends on a couple of configuration settings.

In releases of Hadoop up to and including the 0.20 release series, `mapred.job.tracker` determines the means of execution. If this configuration property is set to `local`, the default, then the local job runner is used. This runner runs the whole job in a single JVM. It's designed for testing and for running MapReduce programs on small datasets.

Alternatively, if `mapred.job.tracker` is set to a colon-separated host and port pair, then the property is interpreted as a jobtracker address, and the runner submits the job to the jobtracker at that address. The whole process in described in detail in the next section.

In Hadoop 0.23.0 a new MapReduce implementation was introduced. The new implementation (called MapReduce 2) is built on a system called YARN, described in "YARN (MapReduce 2)" on page 194. For now, just note that the framework that is used for execution is set by the `mapreduce.framework.name` property, which takes the values `local` (for the local job runner), `classic` (for the "classic" MapReduce frame-

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

work, also called MapReduce 1, which uses a jobtracker and tasktrackers), and `yarn` (for the new framework).

It's important to realize that the old and new MapReduce APIs are not the same thing as the classic and YARN-based MapReduce implementations (MapReduce 1 and 2 respectively). The APIs are user-facing client-side features and determine how you write MapReduce programs, while the implementations are just different ways of running MapReduce programs. All four combinations are supported: both the old and new API run on both MapReduce 1 and 2. Table 1-2 lists which of these combinations are supported in the different Hadoop releases.

Classic MapReduce (MapReduce 1)

A job run in classic MapReduce is illustrated in Figure 6-1. At the highest level, there are four independent entities:

- The client, which submits the MapReduce job.
- The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is `JobTracker`.
- The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is `TaskTracker`.
- The distributed filesystem (normally HDFS, covered in Chapter 3), which is used for sharing job files between the other entities.

Job Submission

The `submit()` method on `Job` creates an internal `JobSummitter` instance and calls `submitJobInternal()` on it (step 1 in Figure 6-1). Having submitted the job, `waitForCompletion()` polls the job's progress once a second and reports the progress to the console if it has changed since the last report. When the job is complete, if it was successful, the job counters are displayed. Otherwise, the error that caused the job to fail is logged to the console.

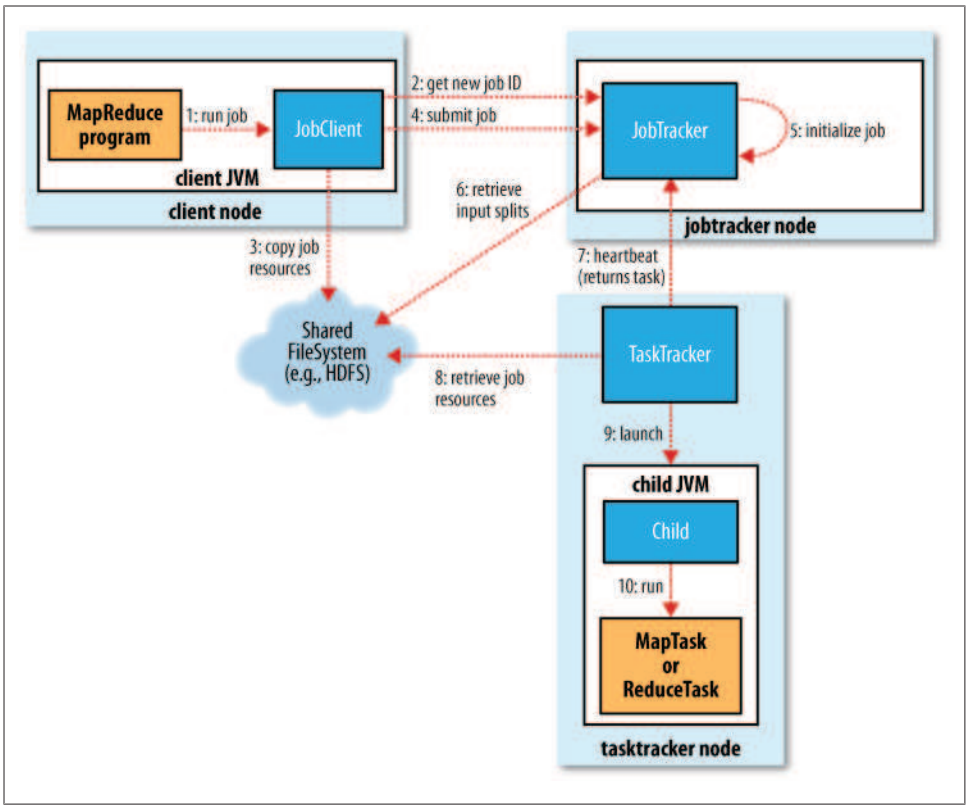

Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

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

- Asks the jobtracker for a new job ID (by calling `getNewJobId()` on `JobTracker`) (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, then the job is not submitted and an error is thrown to the MapReduce program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the `mapred.submit.replication` property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).

- Tells the jobtracker that the job is ready for execution (by calling `submitJob()` on `JobTracker`) (step 4).

Job Initialization

When the `JobTracker` receives a call to its `submitJob()` method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6). It then creates one map task for each split. The number of reduce tasks to create is determined by the `mapred.reduce.tasks` property in the `Job`, which is set by the `setNumReduceTasks()` method, and the scheduler simply creates this number of reduce tasks to be run. Tasks are given IDs at this point.

In addition to the map and reduce tasks, two further tasks are created: a job setup task and a job cleanup task. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to cleanup after all the reduce tasks are complete. The `OutputCommitter` that is configured for the job determines the code to be run, and by default this is a `FileOutputCommitter`. For the job setup task it will create the final output directory for the job and the temporary working space for the task output, and for the job cleanup task it will delete the temporary working space for the task output. The commit protocol is described in more detail in "Output Committers" on page 215.

Task Assignment

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive, but they also double as a channel for messages. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There are various scheduling algorithms as explained later in this chapter (see "Job Scheduling" on page 204), but the default one simply maintains a priority list of jobs. Having chosen a job, the jobtracker now chooses a task for the job.

Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of memory on the tasktracker; see "Memory" on page 305.) The default scheduler fills empty map task slots before reduce task slots, so if the tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.

To choose a reduce task, the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations. For a map task, however, it takes account of the tasktracker's network location and picks a task whose input split is as close as possible to the tasktracker. 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 from the one they are running on. You can tell the proportion of each type of task by looking at a job's counters (see "Built-in Counters" on page 257).

Task Execution

Now that the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk; see "Distributed Cache" on page 288 (step 8). Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Third, it creates an instance of `TaskRunner` to run the task.

`TaskRunner` launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce functions don't affect the tasktracker (by causing it to crash or hang, for example). It is, however, possible to reuse the JVM between tasks; see "Task JVM Reuse" on page 216.

The child process communicates with its parent through the *umbilical* interface. This way it informs the parent of the task's progress every few seconds until the task is complete.

Each task can perform setup and cleanup 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 215). The cleanup action is used to commit the task, which in the case of file-based jobs means that its output is written to the final location for that task. The commit protocol ensures that when speculative execution is enabled ("Speculative Execution" on page 213), only one of the duplicate tasks is committed and the other is aborted.

Streaming and Pipes. Both Streaming and Pipes run special map and reduce tasks for the purpose of launching the user-supplied executable and communicating with it (Figure 6-2).

In the case of Streaming, the Streaming task communicates with the process (which may be written in any language) using standard input and output streams. The Pipes task, on the other hand, listens on a socket and passes the C++ process a port number in its environment, so that on startup, the C++ process can establish a persistent socket connection back to the parent Java Pipes task.

In both cases, 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 tasktracker's point of view, it is as if the tasktracker child process ran the map or reduce code itself.

Progress and Status Updates

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is 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*, that is, 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 205). For example, if the task has run the reducer on half its input, then the task's progress is ⅚, since it has completed the copy and sort phases (⅓ each) and is halfway through the reduce phase (⅙).

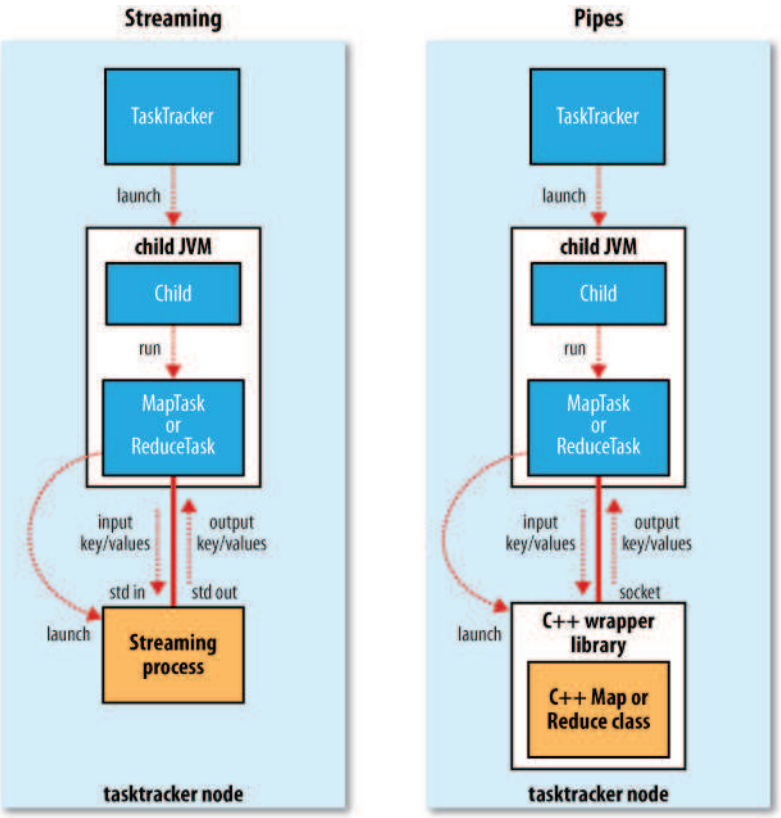

Figure 6-2. The relationship of the Streaming and Pipes executable to the tasktracker and its child

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 though it cannot be expressed as a percentage of the total number that will be written, since the latter figure may not be known, even by the task producing the output.

Progress reporting is important, as it means 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 on a reporter (using `Reporter`'s `setStatus()` method)
- Incrementing a counter (using `Reporter`'s `incrCounter()` method)
- Calling `Reporter`'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 25), either those built into the framework, such as the number of map output records written, or ones defined by users.

If a task reports progress, it sets a flag to indicate that the status change should be sent to the tasktracker. The flag is checked in a separate thread every three seconds, and if set it notifies the tasktracker of the current task status. Meanwhile, the tasktracker is sending heartbeats to the jobtracker every five seconds (this is a minimum, as the heartbeat interval is actually dependent on the size of the cluster: for larger clusters, the interval is longer), and the status of all the tasks being run by the tasktracker is sent in the call. Counters are sent less frequently than every five seconds, because they can be relatively high-bandwidth.

The jobtracker combines these updates to produce a global view of the status of all the jobs being run and their constituent tasks. Finally, as mentioned earlier, the `Job` receives the latest status by polling the jobtracker every second. Clients can also use `Job`'s `getStatus()` method to obtain a `JobStatus` instance, which contains all of the status information for the job.

The method calls are illustrated in Figure 6-3.

Job Completion

When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), 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.

The jobtracker 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 `job.end.notification.url` property.

Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

YARN (MapReduce 2)

For very large clusters in the region of 4000 nodes and higher, the MapReduce system described in the previous section begins to hit scalability bottlenecks, so in 2010 a group at Yahoo! began to design the next generation of MapReduce. The result was *YARN*, short for Yet Another Resource Negotiator (or if you prefer recursive ancronyms, YARN Application Resource Negotiator).[2]

2. You can read more about the motivation for and development of YARN in Arun C Murthy's post, The Next Generation of Apache Hadoop MapReduce.

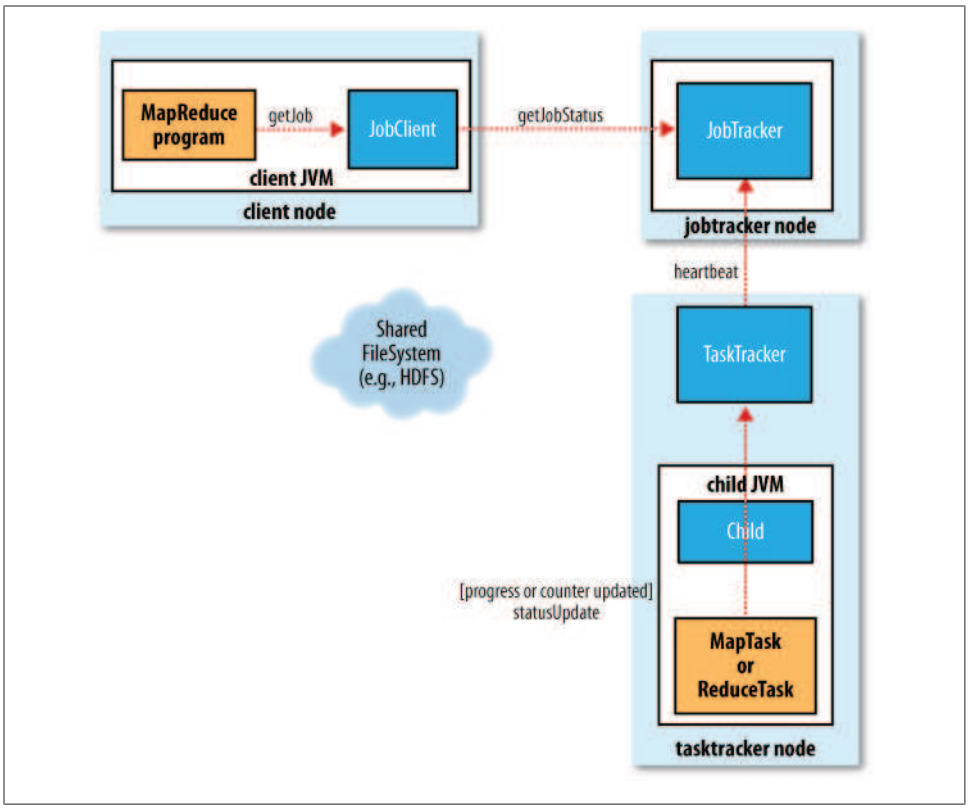

Figure 6-3. How status updates are propagated through the MapReduce 1 system

YARN meets the scalability shortcomings of "classic" MapReduce by splitting the responsibilities of the jobtracker into separate entities. The jobtracker takes care of both job scheduling (matching tasks with tasktrackers) and task progress monitoring (keeping track of tasks and restarting failed or slow tasks, and doing task bookkeeping such as maintaining counter totals).

YARN separates these two roles into two independent daemons: a *resource manager* to manage the use of resources across the cluster, and an *application master* to manage the lifecycle of applications running on the cluster. The idea is that an application master negotiates with the resource manager for cluster resources—described in terms of a number of *containers* each with a certain memory limit—then runs application-specific processes in those containers. The containers are overseen by *node managers* running on cluster nodes, which ensure that the application does not use more resources than it has been allocated.[3]

3. At the time of writing, memory is the only resource that is managed, and node managers will kill any containers that exceed their allocated memory limits.

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.

As described, YARN is more general than MapReduce, and in fact MapReduce is just one type of YARN application. There are a few other YARN applications—such as a distributed shell that can run a script on a set of nodes in the cluster—and others are actively being worked on (some are listed at *http://wiki.apache.org/hadoop/PoweredByYarn*). The beauty of YARN's design is that different YARN applications can co-exist on the same cluster—so a MapReduce application can run at the same time as an MPI application, for example—which brings great benefits for managability and cluster utilization.

Furthermore, 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 managable. (Note that some parts of MapReduce, like the job history server and the shuffle handler, as well as YARN itself, still need to be upgraded across the cluster.)

MapReduce on YARN involves more entities than classic MapReduce. They are:[4]

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

The process of running a job is shown in Figure 6-4, and described in the following sections.

Job Submission

Jobs are submitted in MapReduce 2 using the same user API as MapReduce 1 (step 1). MapReduce 2 has an implementation of `ClientProtocol` that is activated when `mapre`

4. 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), which are a part of the jobtracker and the tasktracker respectively in classic MapReduce, but are independent entities in YARN.

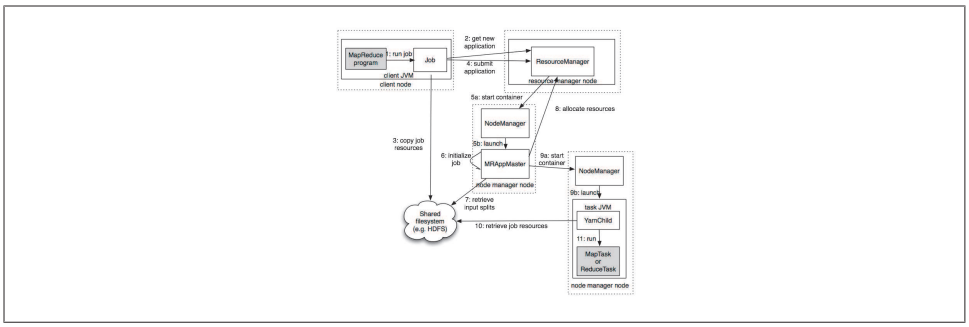

Figure 6-4. How Hadoop runs a MapReduce job using YARN

`duce.framework.name` is set to `yarn`. The submission process is very similar to the classic implementation. The new job ID is retrieved from the resource manager (rather than the jobtracker), although in the nomenclature of YARN it is an application ID (step 2). The job client checks the output specification of the job; computes input splits (although there is an option to generate them on the cluster, `yarn.app.mapreduce.am.compute-splits-in-cluster`, which can be beneficial for jobs with many splits); and copies job resources (including the job JAR, configuration, and split information) to HDFS (step 3). Finally, the job is submitted by calling `submitApplication()` on the resource manager (step 4).

Job Initialization

When the resource manager receives a call to its `submitApplication()`, it hands off the request to the 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, and a number of reduce task objects determined by the `mapreduce.job.reduces` property.

The next thing the application master does is decide how to run the tasks that make up the MapReduce job. If the job is small, the application master may choose to run them in the same JVM as itself, since it judges the overhead of allocating new containers and running tasks in them as outweighing the gain to be had in running them in parallel, compared to running them sequentially on one node. (This is different to MapReduce 1, where small jobs are never run on a single tasktracker.) Such a job is said to be *uberized*, or run as an *uber task*.

What qualifies as a small job? By default one that has less than 10 mappers, only one reducer, and the input size is less than the size of one HDFS block. (These values may

be changed for a job by setting `mapreduce.job.ubertask.maxmaps`, `mapreduce.job.ubertask.maxreduces`, and `mapreduce.job.ubertask.maxbytes`.) It's also possible to disable uber tasks entirely (by setting `mapreduce.job.ubertask.enable` to `false`).

Before any tasks can be run the job setup method is called (for the job's `OutputCommitter`), to create the job's output directory. In contrast to MapReduce 1, where it is called in a special task that is run by the tasktracker, in the YARN implementation the method is called directly by the application master.

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). Each request, which are piggybacked on heartbeat calls, includes information about each map task's data locality, in particular the hosts and corresponding racks that the input split resides on. The scheduler uses this information to make scheduling decisions (just like a jobtracker's scheduler does): it attempts to place tasks on data-local nodes in the ideal case, but if this is not possible the scheduler prefers rack-local placement to non-local placement.

Requests also specify memory requirements for tasks. By default both map and reduce tasks are allocated 1024 MB of memory, but this is configurable by setting `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb`.

The way memory is allocated is different to MapReduce 1, where tasktrackers have a fixed number of "slots", set at cluster configuration time, and each task runs in a single slot. Slots have a maximum memory allowance, which again is fixed for a cluster, and which leads both to problems of under utilization when tasks use less memory (since other waiting tasks are not able to take advantage of the unused memory) and problems of job failure when a task can't complete since it can't get enough memory to run correctly.

In YARN, resources are more fine-grained, so both these problems can be avoided. In particular, applications may request a memory capability that is anywhere between the minimum allocation and a maximum allocation, and which must be a multiple of the minimum allocation. Default memory allocations are scheduler-specific, and for the capacity scheduler the default minimum is 1024 MB (set by `yarn.scheduler.capacity.minimum-allocation-mb`), and the default maximum is 10240 MB (set by `yarn.scheduler.capacity.maximum-allocation-mb`). Thus, tasks can request any memory allocation between 1 and 10 GB (inclusive), in multiples of 1 GB (the scheduler will round to the nearest multiple if needed), by setting `mapreduce.map.memory.mb` and `mapreduce.reduce.memory.mb` appropriately.

Task Execution

Once a task has been assigned a container by the resource manager's scheduler, the application master starts the container by contacting the node manager (steps 9a and

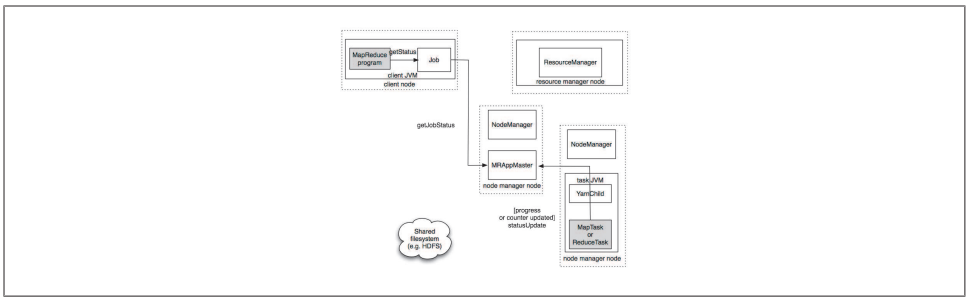

Figure 6-5. How status updates are propagated through the MapReduce 2 system

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). Finally, it runs the map or reduce task (step 11).

The `YarnChild` runs in a dedicated JVM, for the same reason that tasktrackers spawn new JVMs for tasks in MapReduce 1: to isolate user code from long-running system daemons. Unlike MapReduce 1, however, YARN does not support JVM reuse so each task runs in a new JVM.

Streaming and Pipes programs work in the same way as MapReduce 1. The `YarnChild` launches the Streaming or Pipes process and communicates with it using standard input/output or a socket (respectively), as shown in Figure 6-2 (except the child and subprocesses run on node managers, not tasktrackers).

Progress and Status Updates

When running under YARN, the task reports its progress and status (including counters) back to its application master every three seconds (over the umbilical interface), which has an aggregate view of the job. The process is illustrated in Figure 6-5. Contrast this to MapReduce 1, where progress updates flow from the child through the tasktracker to the jobtracker for aggregation.

The client polls the application master every second (set via `mapreduce.client.progressmonitor.pollinterval`) to receive progress updates, which are usually displayed to the user.

Job Completion

As well as polling the application master for progress, every five seconds the client checks whether the job has completed when using the `waitForCompletion()` method on `Job`. The polling interval can be set via the `mapreduce.client.completion.pollinterval` configuration property.

Notification of job completion via an HTTP callback is also supported like in MapReduce 1. In MapReduce 2 the application master initiates the callback.

On job completion the application master and the task containers clean up their working state, and the `OutputCommitter`'s job cleanup 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.

Failures in Classic MapReduce

In the MapReduce 1 runtime there are three failure modes to consider: failure of the running task, failure of the tastracker, and failure of the jobtracker. Let's look at each in turn.

Task Failure

Consider first the case of the child task failing. The most common way that this happens is when user code in the map or reduce task throws a runtime exception. If this happens, the child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as *failed*, freeing up a slot to run 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 child 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 tasktracker notices that the process has exited and marks the attempt as failed.

Hanging tasks are dealt with differently. The tasktracker notices that it hasn't received a progress update for a while and proceeds to mark the task as failed. The child JVM process will be automatically killed after this period.[5] The timeout period after which tasks are considered failed is normally 10 minutes and can be configured on a per-job

5. If a Streaming or Pipes process hangs, the tasktracker will kill it (along with the JVM that launched it) only in one the following circumstances: either `mapred.task.tracker.task-controller` is set to `org.apache.hadoop.mapred.LinuxTaskController`, or the default task controller in being used (`org.apache.hadoop.mapred.DefaultTaskController`) and the `setsid` command is available on the system (so that the child JVM and any processes it launches are in the same process group). In any other case orphaned Streaming or Pipes processes will accumulate on the system, which will impact utilization over time.

basis (or a cluster basis) by setting the `mapred.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 slot, 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 will suffice (see "What Constitutes Progress in MapReduce?" on page 193).

When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call), it will reschedule execution of the task. The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails four times (or more), it will not be retried further. This value is configurable: the maximum number of attempts to run a task is controlled by the `mapred.map.max.attempts` property for map tasks and `mapred.reduce.max.attempts` 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 `mapred.max.map.failures.percent` and `mapred.max.reduce.failures.percent` 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, see "Speculative Execution" on page 213), or because the tasktracker it was running on failed, and the jobtracker 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 `mapred.map.max.attempts` and `mapred.reduce.max.attempts`), since 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 `hadoop job` to see the options). Jobs may also be killed by the same mechanisms.

Tasktracker Failure

Failure of a tasktracker is another failure mode. If a tasktracker fails by crashing, or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently). The jobtracker will notice a tasktracker that has stopped sending heartbeats (if it hasn't received one for 10 minutes, configured via the `mapred.tasktracker.expiry.interval` property, in milliseconds) and remove it from its pool of tasktrackers to schedule tasks on. The jobtracker arranges for map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task. Any tasks in progress are also rescheduled.

A tasktracker can also be *blacklisted* by the jobtracker, even if the tasktracker has not failed. If more than four tasks from the same job fail on a particular tasktracker (set by (`mapred.max.tracker.failures`), then the jobtracker records this as a fault. A tasktracker is blacklisted if the number of faults is over some minimum threshold (four, set by `mapred.max.tracker.blacklists`) and is significantly higher than the average number of faults for tasktrackers in the cluster cluster.

Blacklisted tasktrackers are not assigned tasks, but they continue to communicate with the jobtracker. Faults expire over time (at the rate of one per day), so tasktrackers get the chance to run jobs again simply by leaving them running. Alternatively, if there is an underlying fault that can be fixed (by replacing hardware, for example), the tasktracker will be removed from the jobtracker's blacklist after it restarts and rejoins the cluster.

Jobtracker Failure

Failure of the jobtracker is the most serious failure mode. Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure—so in this case the job fails. However, this failure mode has a low chance of occurring, since the chance of a particular machine failing is low. The good news is that the situation is improved in YARN, since one of its design goals is to eliminate single points of failure in MapReduce.

After restarting a jobtracker, any jobs that were running at the time it was stopped will need to be re-submitted. There is a configuration option that attempts to recover any running jobs (`mapred.jobtracker.restart.recover`, turned off by default), however it is known not to work reliably, so should not be used.

Failures in YARN

For MapReduce programs running on YARN, 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

Failure of the running task is similar to the classic case. Runtime exceptions and sudden exits of the JVM are propagated back to the application master and the task attempt is marked as failed. Likewise, hanging tasks are noticed by the application master by the absence of a ping over the umbilical channel (the timeout is set by `mapreduce.task.timeout`), and again the task attempt is marked as failed.

The configuration properties for determining when a task is considered to be failed are the same as the classic case: a task is marked as failed after four attempts (set by `mapreduce.map.maxattempts` for map tasks and `mapreduce.reduce.maxattempts` for reducer tasks). A job will be failed if more than `mapreduce.map.failures.maxpercent` per-

cent of the map tasks in the job fail, or more than `mapreduce.reduce.failures.maxpercent` percent of the reduce tasks fail.

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 tried multiple times in the event of failure. By default, applications are marked as failed if they fail once, but this can be increased by setting the property `yarn.resourcemanager.am.max-retries`.

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 can recover the state of the tasks that had already been run by the (failed) application so they don't have to be rerun. By default, recovery is not enabled, so failed application masters will not rerun all their tasks, but you can turn it on by setting `yarn.app.mapreduce.am.job.recovery.enable` to `true`.

The client polls the application master for progress reports, so 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 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.

Node Manager Failure

If a node manager fails, then it will stop sending heartbeats to the resource manager, and the node manager will be removed from the resource manager's pool of available nodes. The property `yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms`, which defaults to 600000 (10 minutes), determines the minimum time the resource manager waits before considering a node manager that has sent no heartbeat in that time as failed.

Any task or application master running on the failed node manager will be recovered using the mechanisms described in the previous two sections.

Node managers may be blacklisted if the number of failures for the application is high. 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 threshold may be set with `mapreduce.job.maxtaskfailures.per.tracker`.

Resource Manager Failure

Failure of the resource manager is serious, since without it neither jobs nor task containers can be launched. The resource manager was designed from the outset to be able to recover from crashes, by using a checkpointing mechanism to save its state to persistent storage, although at the time of writing the latest release did not have a complete implementation.

After a crash, a new resource manager instance is brought up (by an adminstrator) and it recovers from the saved state. The state consists of the node managers in the system as well as the running applications. (Note that tasks are not part of the resource manager's state, since they are managed by the application. Thus the amount of state to be stored is much more managable than that of the jobtracker.)

The storage used by the reource manager is configurable via the `yarn.resourceman ager.store.class` property. The default is `org.apache.hadoop.yarn.server.resource manager.recovery.MemStore`, which keeps the store in memory, and is therefore not highly-available. However, there is a ZooKeeper-based store in the works that will support reliable recovery from resource manager failures in the future.

Job Scheduling

Early versions of Hadoop had a very simple approach to scheduling users' jobs: they ran in order of submission, using a FIFO scheduler. Typically, each job would use the whole cluster, so jobs had to wait their turn. Although a shared cluster offers great potential for offering large resources to many users, the problem of sharing resources fairly between users requires a better scheduler. Production jobs need to complete in a timely manner, while allowing users who are making smaller ad hoc queries to get results back in a reasonable time.

Later on, the ability to set a job's priority was added, via the `mapred.job.priority` property or the `setJobPriority()` method on `JobClient` (both of which take one of the values `VERY_HIGH`, `HIGH`, `NORMAL`, `LOW`, `VERY_LOW`). When the job scheduler is choosing the next job to run, it selects one with the highest priority. However, with the FIFO scheduler, priorities do not support *preemption*, so a high-priority job can still be blocked by a long-running low priority job that started before the high-priority job was scheduled.

MapReduce in Hadoop comes with a choice of schedulers. The default in MapReduce 1 is the original FIFO queue-based scheduler, and there are also multiuser schedulers called the Fair Scheduler and the Capacity Scheduler.

MapReduce 2 comes with the Capacity Scheduler (the default), and the FIFO scheduler.

The Fair Scheduler

The Fair Scheduler aims to give every user a fair share of the cluster capacity over time. If a single job is running, it gets all of the cluster. As more jobs are submitted, free task slots are given to the jobs in such a way as to give each user a fair share of the cluster. A short job belonging to one user will complete in a reasonable time even while another user's long job is running, and the long job will still make progress.

Jobs are placed in pools, and by default, each user gets their own pool. A user who submits more jobs than a second user will not get any more cluster resources than the second, on average. It is also possible to define custom pools with guaranteed minimum capacities defined in terms of the number of map and reduce slots, and to set weightings for each pool.

The Fair Scheduler supports preemption, so if a pool has not received its fair share for a certain period of time, then the scheduler will kill tasks in pools running over capacity in order to give the slots to the pool running under capacity.

The Fair Scheduler is a "contrib" module. To enable it, place its JAR file on Hadoop's classpath, by copying it from Hadoop's *contrib/fairscheduler* directory to the *lib* directory. Then set the `mapred.jobtracker.taskScheduler` property to:

```
org.apache.hadoop.mapred.FairScheduler
```

The Fair Scheduler will work without further configuration, but to take full advantage of its features and how to configure it (including its web interface), refer to README in the *src/contrib/fairscheduler* directory of the distribution.

The Capacity Scheduler

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

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*.[6] In this section, we look at how the shuffle works, as a basic understanding would be helpful, should you need to optimize a Map-

Reduce 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 (and may change over time, this is for version 0.20). 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 6-6 shows what happens.

Each map task has a circular memory buffer that it writes the output to. The buffer is 100 MB by default, a size which can be tuned by changing the `io.sort.mb` property. When the contents of the buffer reaches a certain threshold size (`io.sort.spill.percent`, default `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, the map will block until the spill is complete.

Spills are written in round-robin fashion to the directories specified by the `mapred.local.dir` property, in a job-specific subdirectory.

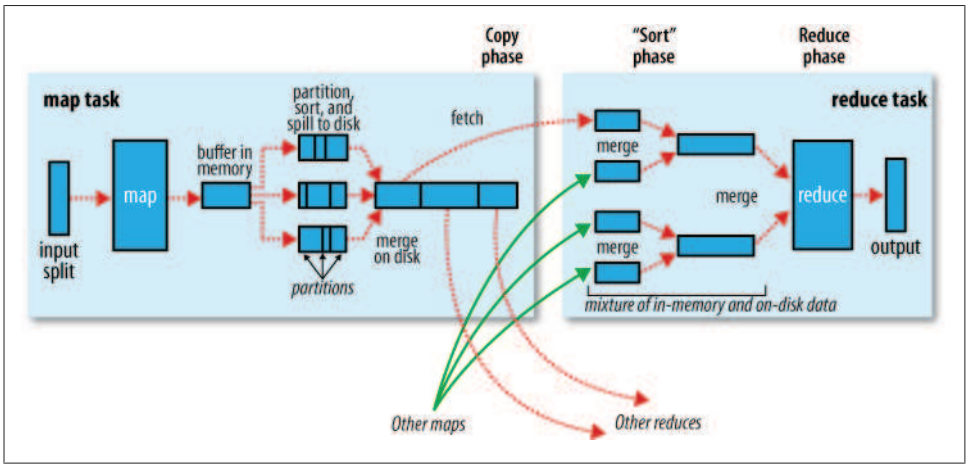

Figure 6-6. Shuffle and sort in MapReduce

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

6. 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.

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 `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 `min.num.spills.for.combine` property) then 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, then 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, since 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 by setting `mapred.compress.map.output` to `true`. The compression library to use is specified by `mapred.map.output.compression.codec`; see "Compression" on page 85 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 `tasktracker.http.threads` property—this setting is per tasktracker, not per map task slot. The default of 40 may need increasing for large clusters running large jobs. In MapReduce 2, this property is not applicable since the maximum number of threads used is set automatically based on the number of processors on the machine. (MapReduce 2 uses Netty, which by default allows up to twice as many threads as there are processors.)

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. Furthermore, 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 `mapred.reduce.parallel.copies` property.

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

As map tasks complete successfully, they notify their parent tasktracker of the status update, which in turn notifies the jobtracker. (In MapReduce 2, the tasks notify their application master directly.) These notifications are transmitted over the heartbeat communication mechanism described earlier. Therefore, for a given job, the jobtracker (or 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 jobtracker (or application master), which is after the job has completed.

The map outputs are copied to the reduce task JVM's memory if they are small enough (the buffer's size is controlled by `mapred.job.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 `mapred.job.shuffle.merge.percent`), or reaches a threshold number of map outputs (`mapred.inmem.merge.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 `io.sort.factor` property, just like in the map's merge), then there would be 5 rounds. Each round would merge 10 files into one, so at the end there would be five 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 6-7.

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.

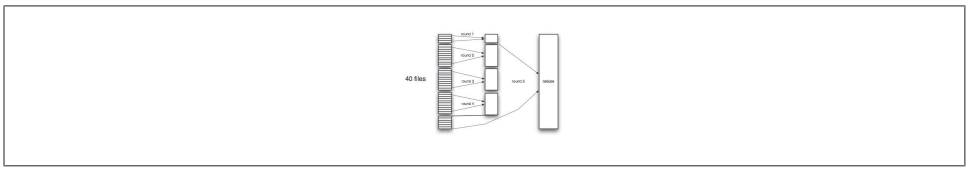

Figure 6-7. Efficiently merging 40 file segments with a merge factor of 10

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, since the tasktracker node (or 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 6-1 and 6-2, 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 (by avoiding 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" on page 305 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, then you can set the `io.sort.*` properties appropriately to minimize the number of spills. In particular,

you should increase `io.sort.mb` if you can. There is a MapReduce counter ("Spilled records"; see "Counters" on page 257) 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. By default, this does not happen, since for the general case all the memory is reserved for the reduce function. But if your reduce function has light memory requirements, then setting `mapred.inmem.merge.threshold` to `0` and `mapred.job.reduce.input.buffer.percent` to `1.0` (or a lower value; see Table 6-2) may bring a performance boost.

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 315).

In April 2008, Hadoop won the general-purpose terabyte sort benchmark (described in "TeraByte Sort on Apache Hadoop" on page 601), and one of the optimizations used was this one of keeping the intermediate data in memory on the reduce side.

Table 6-1. Map-side tuning properties

Property name	Type	Default value	Description
`io.sort.mb`	`int`	`100`	The size, in megabytes, of the memory buffer to use while sorting map output.
`io.sort.record.percent`	`float`	`0.05`	The proportion of `io.sort.mb` reserved for storing record boundaries of the map outputs. The remaining space is used for the map output records themselves. This property was removed in release 0.21.0 as the shuffle code was improved to do a better job of using all the available memory for map output and accounting information.
`io.sort.spill.percent`	`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.
`io.sort.factor`	`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.

Property name	Type	Default value	Description
min.num.spills.for.combine	int	3	The minimum number of spill files needed for the combiner to run (if a combiner is specified).
mapred.compress.map.output	boolean	false	Compress map outputs.
mapred.map.output.compression.codec	Class name	org.apache.hadoop.io.compress.DefaultCodec	The compression codec to use for map outputs.
task tracker.http.threads	int	40	The number of worker threads per tasktracker for serving the map outputs to reducers. This is a cluster-wide setting and cannot be set by individual jobs. Not applicable in MapReduce 2.

Table 6-2. Reduce-side tuning properties

Property name	Type	Default value	Description
mapred.reduce.parallel.copies	int	5	The number of threads used to copy map outputs to the reducer.
mapred.reduce.copy.backoff	int	300	The maximum amount of time, in seconds, to spend retrieving one map output for a reducer before declaring it as failed. The reducer may repeatedly reattempt a transfer within this time if it fails (using exponential backoff).
io.sort.factor	int	10	The maximum number of streams to merge at once when sorting files. This property is also used in the map.
mapred.job.shuffle.input.buffer.percent	float	0.70	The proportion of total heap size to be allocated to the map outputs buffer during the copy phase of the shuffle.
mapred.job.shuffle.merge.percent	float	0.66	The threshold usage proportion for the map outputs buffer (defined by mapred.job.shuffle.input.buffer.percent) for starting the process of merging the outputs and spilling to disk.
mapred.inmem.merge.threshold	int	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 mapred.job.shuffle.merge.percent.
mapred.job.reduce.input.buffer.percent	float	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

Property name	Type	Default value	Description
			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 the chapter in "Anatomy of a MapReduce Job Run" on page 187. 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 239), and a map or reduce task can find out the attempt number of the task. The properties in Table 6-3 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 6-3. Task environment properties

Property name	Type	Description	Example
`mapred.job.id`	`String`	The job ID. (See "Job, Task, and Task Attempt IDs" on page 163 for a description of the format.)	`job_200811201130_0004`
`mapred.tip.id`	`String`	The task ID.	`task_200811201130_0004_m_000003`
`mapred.task.id`	`String`	The task attempt ID. (*Not* the task ID.)	`attempt_200811201130_0004_m_000003_0`
`mapred.task.partition`	`int`	The index of the task within the job.	`3`
`mapred.task.is.map`	`boolean`	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 `mapred.job.id` property from within a Python Streaming script:

```
os.environ["mapred_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 otherwise be if the tasks ran sequentially. This makes 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 mis-configuration, but the causes may be hard to detect since 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, a speculative task is launched only after all the tasks for a job have been launched, and then only for tasks that have been running for some time (at least a minute) and have failed to make as much progress, on average, as the other tasks from the job. 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, then the speculative task is killed; on the other hand, if the speculative task finishes first, then the original is killed.

Speculative execution is an optimization, not a feature to make jobs run more reliably. If there are bugs that sometimes cause a task to hang or slow down, then 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 6-4.

Table 6-4. Speculative execution properties

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

Why would you ever want to turn off speculative execution? 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 that speculative execution is turned off is for tasks that are not idempotent. 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, and this 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 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, `${mapred.output.dir}`, and a temporary working space for task output, `${mapred.output.dir}/_temporary`.

If the job succeeds then 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, then the `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, since 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 com-

mit 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 then `commitTask()` is called, which in the default implementation moves the temporaray task output directory (which has the task attempt ID in its name to avoid conflicts between task attempts) to the final output path, `${mapred.output.dir}`. 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, and the others will be aborted. This suitation 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. Another case is if two task attempts were running concurrently as speculative duplicates, then 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 the `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, like HDFS. (There are other ways to produce multiple outputs, too, as described in "Multiple Outputs" on page 251.)

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, then the side files for tasks that successfully complete will be promoted to the output directory automatically, while failed tasks will have their side files deleted.

A task may find its working directory by retrieving the value of the `mapred.work.output.dir` property from its configuration file. 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 245). If a map task writes the converted images into its working directory, then they will be promoted to the output directory when the task successfully finishes.

Task JVM Reuse

Hadoop runs tasks in their own Java Virtual Machine to isolate them from other running tasks. The overhead of starting a new JVM for each task can take around a second, which for jobs that run for a minute or so is insignificant. However, jobs that have a

large number of very short-lived tasks (these are usually map tasks), or that have lengthy initialization, can see performance gains when the JVM is reused for subsequent tasks.[7]

Note that, with task JVM reuse enabled, tasks are *not* run concurrently in a single JVM; rather, the JVM runs tasks sequentially. Tasktrackers can, however, run more than one task at a time, but this is always done in separate JVMs. The properties for controlling the tasktrackers' number of map task slots and reduce task slots are discussed in "Memory" on page 305.

The property for controlling task JVM reuse is `mapred.job.reuse.jvm.num.tasks`: it specifies the maximum number of tasks to run for a given job for each JVM launched; the default is 1 (see Table 6-5). No distinction is made between map or reduce tasks, however tasks from different jobs are always run in separate JVMs. The method `setNumTasksToExecutePerJvm()` on `JobConf` can also be used to configure this property.

Table 6-5. Task JVM Reuse properties

Property name	Type	Default value	Description
`mapred.job.reuse.jvm.num.tasks`	`int`	`1`	The maximum number of tasks to run for a given job for each JVM on a tasktracker. A value of `-1` indicates no limit: the same JVM may be used for all tasks for a job.

Tasks that are CPU-bound may also benefit from task JVM reuse by taking advantage of runtime optimizations applied by the HotSpot JVM. After running for a while, the HotSpot JVM builds up enough information to detect performance-critical sections in the code and dynamically translates the Java byte codes of these hot spots into native machine code. This works well for long-running processes, but JVMs that run for seconds or a few minutes may not gain the full benefit of HotSpot. In these cases, it is worth enabling task JVM reuse.

Another place where a shared JVM is useful is for sharing state between the tasks of a job. By storing reference data in a static field, tasks get rapid access to the shared data.

Skipping Bad Records

Large datasets are messy. They often have corrupt records. They often have records that are in a different format. They often have missing fields. In an ideal world, your code would cope gracefully with all of these conditions. In practice, it is often expedient to ignore the offending records. Depending on the analysis being performed, if only a small percentage of records are affected, then skipping them may not significantly affect the result. However, if a task trips up when it encounters a bad record—by throwing a runtime exception—then the task fails. Failing tasks are retried (since the failure may be due to hardware failure or some other reason outside the task's control), but if a

7. JVM reuse is not supported in MapReduce 2.

task fails four times, then the whole job is marked as failed (see "Task Failure" on page 200). If it is the data that is causing the task to throw an exception, rerunning the task won't help, since it will fail in exactly the same way each time.

If you are using `TextInputFormat` ("TextInputFormat" on page 244), then 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 `mapred.linerecordreader.maxlength` to a value in bytes that fits in memory (and is comfortably greater than the length of lines in your input data), the record reader will skip the (long) corrupt lines without the task failing.

The best way to handle corrupt records is in your mapper or reducer code. You can detect the bad record and ignore it, or you can abort the job by throwing an exception. You can also count the total number of bad records in the job using counters to see how widespread the problem is.

In rare cases, though, you can't handle the problem because there is a bug in a third-party library that you can't work around in your mapper or reducer. In these cases, you can use Hadoop's optional *skipping mode* for automatically skipping bad records.[8]

When skipping mode is enabled, tasks report the records being processed back to the tasktracker. When the task fails, the tasktracker retries the task, skipping the records that caused the failure. Because of the extra network traffic and bookkeeping to maintain the failed record ranges, skipping mode is turned on for a task only after it has failed twice.

Thus, for a task consistently failing on a bad record, the tasktracker runs the following task attempts with these outcomes:

1. Task fails.
2. Task fails.
3. Skipping mode is enabled. Task fails, but failed record is stored by the tasktracker.
4. Skipping mode is still enabled. Task succeeds by skipping the bad record that failed in the previous attempt.

Skipping mode is off by default; you enable it independently for map and reduce tasks using the `SkipBadRecords` class. It's important to note that skipping mode can detect only one bad record per task attempt, so this mechanism is appropriate only for detecting occasional bad records (a few per task, say). You may need to increase the maximum number of task attempts (via `mapred.map.max.attempts` and

8. Skipping mode is not supported in the new MapReduce API. See *https://issues.apache.org/jira/browse/MAPREDUCE-1932*.

`mapred.reduce.max.attempts`) to give skipping mode enough attempts to detect and skip all the bad records in an input split.

Bad records that have been detected by Hadoop are saved as sequence files in the job's output directory under the *_logs/skip* subdirectory. These can be inspected for diagnostic purposes after the job has completed (using `hadoop fs -text`, for example).