CHAPTER 7
How MapReduce Works

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

Anatomy of a MapReduce Job Run

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

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

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

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

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

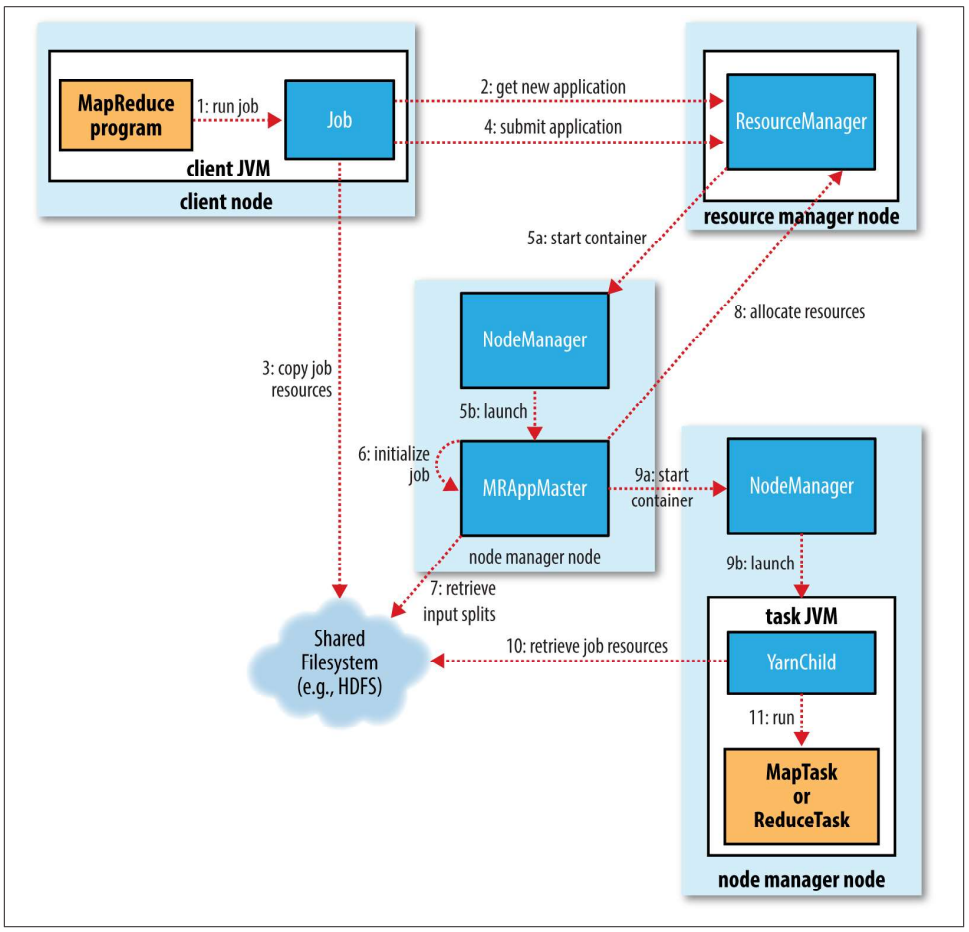

Figure 7-1. How Hadoop runs a MapReduce job

Job Submission

The `submit()` method on `Job` creates an internal `JobSubmitter` instance and calls `submitJobInternal()` on it (step 1 in Figure 7-1). Having submitted the job, `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.

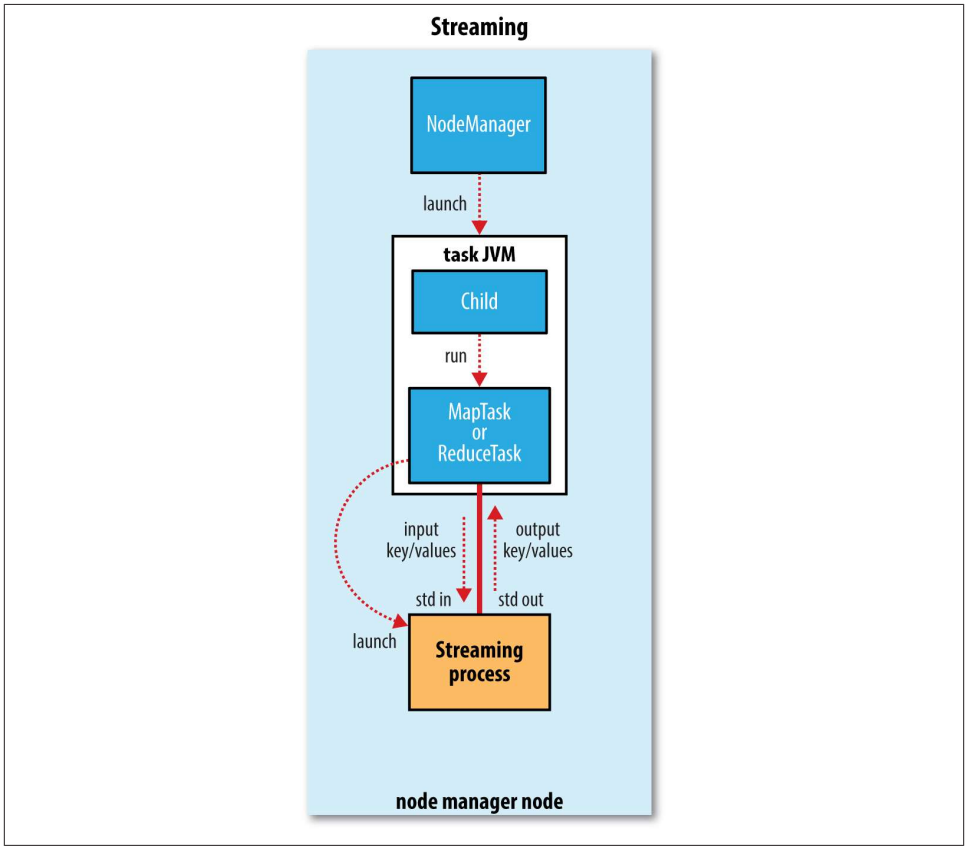

Figure 7-2. The relationship of the Streaming executable to the node manager and the task container

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.

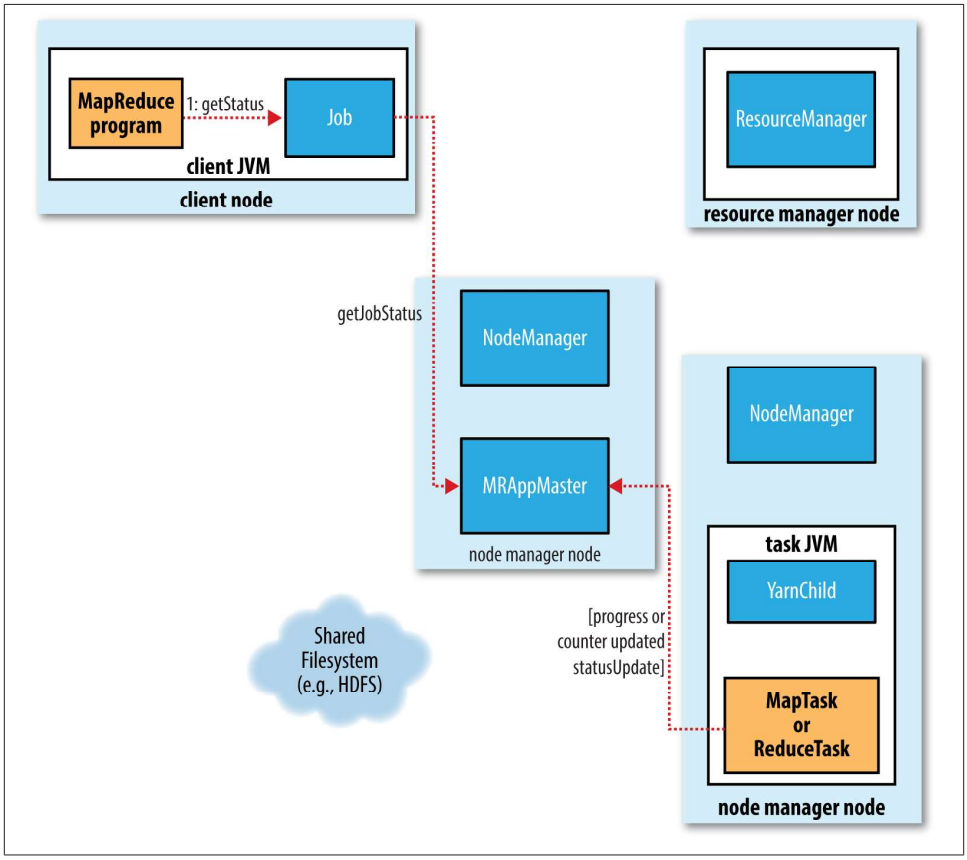

Figure 7-3. How status updates are propagated through the MapReduce system

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 jobtracker in MapReduce 1.)

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

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

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

Shuffle and Sort

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

The Map Side

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

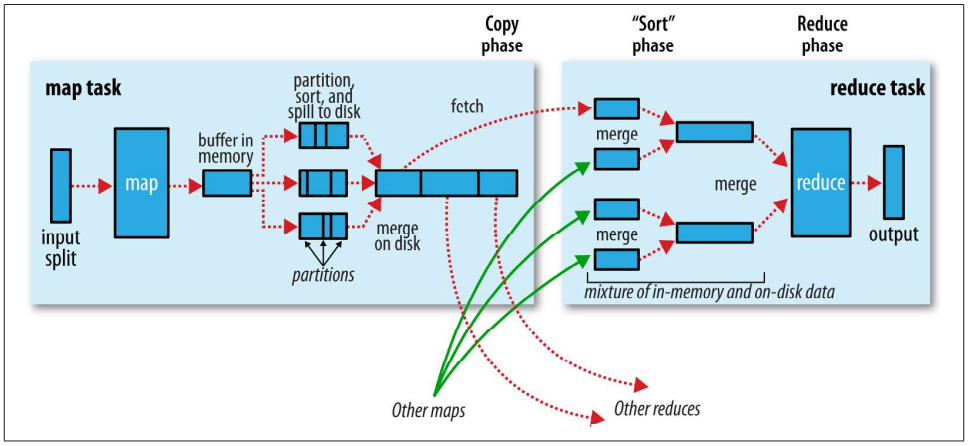

Figure 7-4. Shuffle and sort in MapReduce

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

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

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

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

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

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

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

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

The Reduce Side

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

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

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

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

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

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

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

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

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

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

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

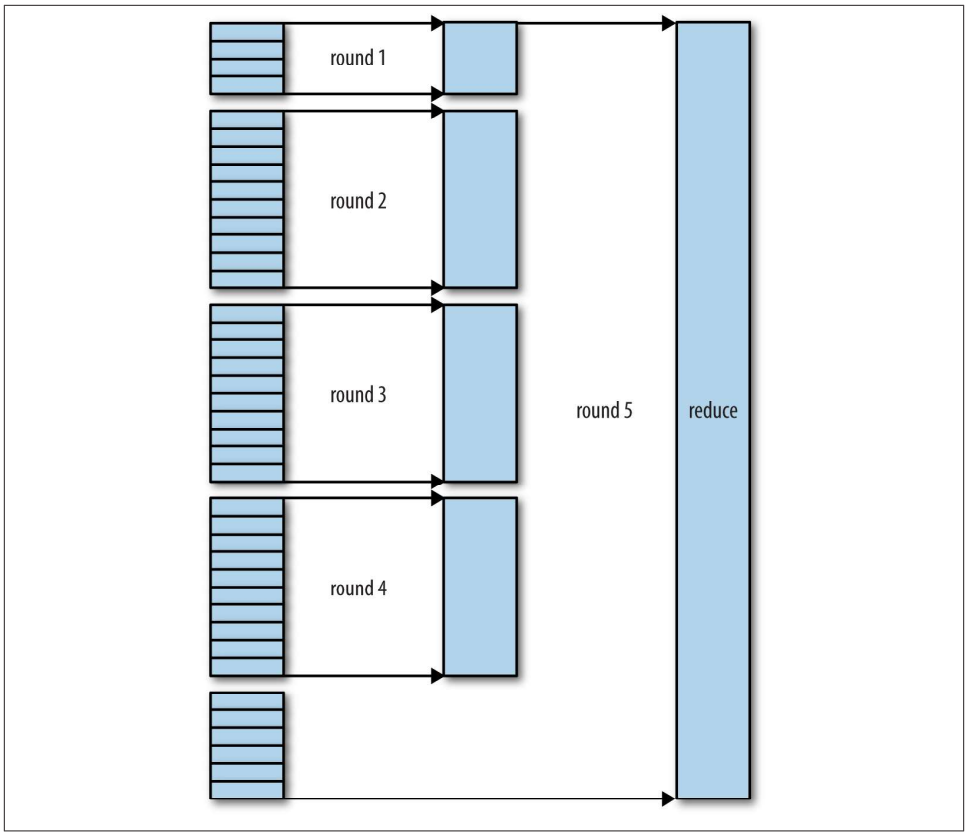

Figure 7-5. Efficiently merging 40 file segments with a merge factor of 10

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

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

Configuration Tuning

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

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

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

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

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

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

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

Table 7-1. Map-side tuning properties

Property name	Type	Default value	Description
`mapreduce.task.io.sort.mb`	`int`	100	The size, in megabytes, of the memory buffer to use while sorting map output.
`mapreduce.map.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.
`mapreduce.task.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.
`mapreduce.map.combine.minspills`	`int`	3	The minimum number of spill files needed for the combiner to run (if a combiner is specified).
`mapreduce.map.output.compress`	`boolean`	`false`	Whether to compress map outputs.
`mapreduce.map.output.compress.codec`	`Classname`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for map outputs.
`mapreduce.shuffle.max.threads`	`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
`mapreduce.reduce.shuffle.parallelcopies`	`int`	5	The number of threads used to copy map outputs to the reducer.
`mapreduce.reduce.shuffle.maxfetchfailures`	`int`	10	The number of times a reducer tries to fetch a map output before reporting the error.
`mapreduce.task.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.
`mapreduce.reduce.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.
`mapreduce.reduce.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.

Property name	Type	Default value	Description
mapreduce.reduce.merge.inmem.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 mapreduce.reduce.shuffle.merge.percent.
mapreduce.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 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
mapreduce.job.id	String	The job ID (see "Job, Task, and Task Attempt IDs" on page 164 for a description of the format)	job_200811201130_0004
mapreduce.task.id	String	The task ID	task_200811201130_0004_m_000003
mapreduce.task.attempt.id	String	The task attempt ID	attempt_200811201130_0004_m_000003_0

Property name	Type	Description	Example
`mapreduce.task.partition`	`int`	The index of the task within the job	`3`
`mapreduce.task.ismap`	`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 `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
`mapreduce.map.speculative`	`boolean`	`true`	Whether extra instances of map tasks may be launched if a task is making slow progress
`mapreduce.reduce.speculative`	`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`	The `Speculator` class implementing the speculative execution policy (MapReduce 2 only)
`yarn.app.mapreduce.am.job.task.estimator.class`	`Class`	`org.apache.hadoop.mapreduce.v2.app.speculate.LegacyTaskRuntimeEstimator`	An implementation of `TaskRuntimeEstimator` 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.