CHAPTER 5

MapReduce Details for Multimachine Clusters

Organizations run Hadoop Core to provide MapReduce services for their processing needs. They may have datasets that can't fit on a single machine, have time constraints that are impossible to satisfy with a small number of machines, or need to rapidly scale the computing power applied to a problem due to varying input set sizes. You will have your own unique reasons for running MapReduce applications.

To do your job effectively, you need to understand all of the moving parts of a MapReduce cluster and of the Hadoop Core MapReduce framework. This chapter will raise the hood and show you some schematics of the engine. This chapter will also provide examples that you can use as the basis for your own MapReduce applications.

Requirements for Successful MapReduce Jobs

For your MapReduce jobs to be successful, the mapper must be able to ingest the input and process the input record, sending forward the records that can be passed to the reduce task or to the final output directly, if no reduce step is required. The reducer must be able to accept the key and value groups that passed through the mapper, and generate the final output of this MapReduce step.

The job must be configured with the location and type of the input data, the mapper class to use, the number of reduce tasks required, and the reducer class and I/O types.

The TaskTracker service will actually run your map and reduce tasks, and the JobTracker service will distribute the tasks and their input split to the various trackers.

The cluster must be configured with the nodes that will run the TaskTrackers, and with the number of TaskTrackers to run per node. The TaskTrackers need to be configured with the JVM parameters, including the classpath for both the TaskTracker and the JVMs that will execute the individual tasks.

There are three levels of configuration to address to configure MapReduce on your cluster. From the bottom up, you need to configure the machines, the Hadoop MapReduce framework, and the jobs themselves.

We'll get started with these requirements by exploring how to launch your MapReduce jobs.

■ **Tip** A Hadoop job is usually part of a production application, which may have many steps, some of which are MapReduce jobs. Hadoop Core, as of version 0.19.0, provides a way of optimizing the data flows between a set of sequential MapReduce jobs. This framework for descriptively and efficiently running sequential MapReduce jobs together is called *chaining*, and uses the `ChainMapper` and the `ChainReducer`, as discussed in Chapter 8. An alternative is the cascading package, available from `http://www.cascading.org/`.

Launching MapReduce Jobs

Jobs within a MapReduce cluster can be launched by constructing a `JobConf` object (details on the `JobConf` object are provided in this book's appendix) and passing it to a `JobClient` object:

```
JobConf conf = new JobConf(MyClass.class);
/** Configuration setup deleted for clarity*/
/** Launch the Job by submitting it to the Framework. */
RunningJob job = JobClient.runJob(conf);
```

You can launch the preceding example from the command line as follows:

```
> bin/hadoop [-libjars jar1.jar,jar2.jar,jar3.jar] jar myjar.jar MyClass
```

The optional `-libjars jar1.jar...` specifications add JARs for your job. The assumption is that `MyClass` is in the `myjar.jar`.

For this to be successful requires a considerable amount of runtime environment setup. Hadoop Core provides a shell script, `bin/hadoop`, which manages the setup for a job. Using this script is the standard and recommended way to start a MapReduce job. This script sets up the process environment correctly for the installation, including inserting the Hadoop JARs and Hadoop configuration directory into the classpath, and launches your application. This behavior is triggered by providing the initial command-line argument `jar` to the `bin/hadoop` script.

Hadoop Core provides several mechanisms for setting the classpath for your application:

- You can set up a fixed base classpath by altering `hadoop-env.sh`, via the `HADOOP_CLASSPATH` environment variable (on all of your machines) or by setting that environment variable in the runtime environment for the user that starts the Hadoop servers.
- You may run your jobs via the `bin/hadoop jar` command and supply a `-libjars` argument with a list of JARs.
- The `DistributedCache` object provides a way to add files or archives to your runtime classpath.

Tip The `mapred.child.java.opts` variable may also be used to specify non-classpath parameters to the child JVMs. In particular, the `java.library.path` variable specifies the path for shared libraries if your application uses the Java Native Interface (JNI). If your application alters the job configuration parameter `mapred.child.java.opts`, it is important to ensure that the JVM memory settings are reset or still present, or your tasks may fail with out-of-memory exceptions.

The advantage of using the `DistributedCache` and `-libjars` is that resources, such as JAR files, do not have to already exist on the TaskTracker nodes. The disadvantages are that the resources must be unpacked on each machine and it is harder to verify which versions of the resources are used.

When launching an application, a number of command-line parameters may be provided. Table 5-1 lists some common command-line arguments. The class `org.apache.hadoop.util.GenericOptionsParser` actually handles the processing of Table 5-1 arguments.

Table 5-1. *Hadoop Standard Command-Line Arguments*

Flag	Description
`-libjars`	A comma-separated list of JAR files to add to the classpath to the job being launched and to the map and reduce tasks run by the TaskTrackers. These JAR files will be staged into HDFS if needed and made available as local files in a temporary job area on each of the TaskTracker nodes.
`-archives`	A comma-separated list of archive files to make available to the running tasks via the distributed cache. These archives will be staged into HDFS if needed.
`-files`	A comma-separated list of files to make available to the running tasks via the distributed cache. These files will be staged into HDFS if needed.
`-fs`	Override the configuration default file system with the supplied URL, the parameter `fs.default.name`.
`-jt`	Override the configuration default JobTracker with the supplied `host port`, the parameter `mapred.job.tracker`.
`-conf`	Use this configuration in place of the `conf/hadoop-default.xml` and `conf/hadoop-site.xml files`.
`-D`	Supply an additional job configuration property in *`key=value`* format. This argument may be provided multiple times. There must be whitespace between the `-D` and the *`key=value`*.

You can use `hadoop jar` to launch an application, as follows:

```
hadoop jar [-fs hdfs://host:port] [-jt host:port] [-conf hadoop-config.xml] ➥
 [-D prop1=value] [-D prop2=value…] [-libjars jar1[,jar2,jar3]] ➥
 [-files file1[,file2,file3]] [-archives archive1[,archive2,archive3]] ➥
applicationjar [main class if not supplied in jar] [arguments to main…]
```

When `hadoop jar` is used, the main method of `org.apache.hadoop.mapred.JobShell` is invoked by the JVM, with all of the remaining command-line arguments. The `JobShell` in turn uses the class `org.apache.hadoop.util.GenericOptionsParser` to process the arguments, as described in Table 5-1.

There are two distinct steps in the argument processing of jobs submitted by the `bin/hadoop` script. The first step is provided by the framework via the `JobShell`. The arguments after `jar` are processed by the `JobShell`, per Table 5-1. The first argument not in the set recognized by the `JobShell` must be the path to a JAR file, which is the job JAR file. If the job JAR file contains a main class specification in the manifest, that class will be the main class called after the first step of argument processing is complete. If the JAR file does not have a main class in the manifest, the next argument becomes required, and is used as main class name. Any remaining unprocessed arguments are passed to the main method of the main class as the arguments. The second step is the processing of the remaining command-line arguments by the user-specified main class.

Using Shared Libraries

Jobs sometime require specific shared libraries. For example, one of my jobs required a shared library that handled job-specific image processing. You can handle this in two ways:

- Pass the shared library via the `DistributedCache` object. For example, using the command-line options `-file libMyStuff.so` would make `libMyStuff.so` available in the current working directory of each task. (The `DistributedCache` object is discussed shortly, in the "Using the Distributed Cache" section.)
- Install the shared library on every TaskTracker machine, and have the JVM library loader path `java.library.path` include the installation directory. The task JVM working directory is part of the `java.library.path` for a task, and any file that is symbolic-linked may be loaded by the JVM.

Caution If you are manually loading shared libraries, the library name passed to `System.loadLibrary()` must not have the trailing `.so`. `System.loadLibrary()` first calls `System.mapLibraryName()` and attempts to load the results. This can result in library load failures that are hard to diagnose.

MapReduce-Specific Configuration for Each Machine in a Cluster

For simplicity and ease of ongoing maintenance, this section assumes identical Hadoop Core installations will be placed on each of the machines, in the same location. The cluster-level configuration is covered in Chapter 3.

The following are the MapReduce-specific configuration requirements for each machine in the cluster:

- You need to install any standard JARs that your application uses, such as Spring, Hibernate, HttpClient, Commons Lang, and so on.
- It is probable that your applications will have a runtime environment that is deployed from a configuration management application, which you will also need to deploy to each machine.
- The machines will need to have enough RAM for the Hadoop Core services plus the RAM required to run your tasks.
- The `conf/slaves` file should have the set of machines to serve as TaskTracker nodes. You may manually start individual TaskTrackers by running the command `bin/hadoop-daemon.sh start tasktracker`, but this is not a recommended practice for starting a cluster.

The `hadoop-env.sh` script has a section for providing custom JVM parameters for the different Hadoop Core servers, including the JobTracker and TaskTrackers. As of Hadoop 0.19.0, the classpath settings are global for all servers. The `hadoop-env.sh` file may be modified and distributed to the machines in the cluster, or the environment variable `HADOOP_JOBTRACKER_OPTS` may be set with JVM options before starting the cluster via the `bin/start-all.sh` command or `bin/start-mapred.sh` command. The environment variable `HADOOP_TASKTRACKER_OPTS` may be set to provide per TaskTracker JVM options. It is much better to modify the file, as the changes are persistent and stored in a single Hadoop-specific location.

When starting the TaskTrackers via the `start-*.sh` scripts, the environment variable `HADOOP_TASKTRACKER_OPTS` may be set in the `hadoop-env.sh` file in the MapReduce `conf` directory on the TaskTracker nodes, or the value may be set in the login shell environment so that the value is present in the environment of commands started via `ssh`. The `start-*.sh` scripts will `ssh` to each target machine, and then run the `bin/hadoop-daemon.sh start tasktracker` command.

Using the Distributed Cache

The `DistributedCache` object provides a programmatic mechanism for specifying the resources needed by the mapper and reducer. The job is actually already using the `DistributedCache` object to a limited degree, if the job creates the `JobConf` object with a class as an argument: `new JobConf(MyMapper.class)`. You may also invoke your MapReduce program using the `bin/hadoop` script and provide arguments for `-libjars`, `-files`, or `-archives`.

The downloadable code for this book (available from this book's details page on the Apress web site, `http://www.apress.com`) includes several source files for the `DistributedCache` examples: `Utils.java`, `DistributedCacheExample.java`, and `DistributedCacheMapper.java`.

■ **Caution** The paths and URIs for `DistributedCache` items are stored as comma-separated lists of strings in the configuration. Any comma characters in the paths will result in unpredictable and incorrect behavior.

Adding Resources to the Task Classpath

Four methods add elements to the Java classpath for the map and reduce tasks. The first three in the following list add archives to the classpath. The archives are unpacked in the job local directory of the task. You can use the following methods to add resources to the task classpath:

`JobConf.setJar(String jar)`: Sets the user JAR for the MapReduce job. It is on the `JobConf` object, but it manipulates the same configuration keys as the `DistributedCache`. The file `jar` will be found, and if necessary, copied into the shared file system, and the full path name on the shared file system stored under the configuration key `mapred.jar`.

`JobConf.setJarByClass(Class cls)`: Determines the JAR that contains the class `cls` and calls `JobConf.setJar(jar)` with that JAR.

`DistributedCache.addArchiveToClassPath(Path archive, Configuration conf)`: Adds an archive path to the current set of classpath entries. This is a static method, and the archive (a zip or JAR file) will be made available to the running tasks via the classpath of the JVM. The archive is also added to the list of cached archives. The contents will be unpacked in the local job directory on each TaskTracker node. The archive path is stored in the configuration under the key `mapred.job.classpath.archives`, and the URI constructed from `archive.makeQualified(conf).toUri()` is stored under the key `mapred.job.classpath.archives`. If the path component of the URI does not exactly equal `archive`, `archive` will not be placed in the classpath of the task correctly.

Caution The `archive` path must be on the JobTracker shared file system, and must be an absolute path. Only the path `/user/hadoop/myjar.jar` is correct; `hdfs://host:8020/user/hadoop/myjar.jar` will fail, as will `hadoop/myjar.jar` or `myjar.jar`.

`DistributedCache.addFileToClassPath(Path file, Configuration conf)`: Adds a file path to the current set of classpath entries. It adds the file to the cache as well. This is a static method that makes the file available to the running tasks via the classpath of the JVM. The file path is stored under the configuration key `mapred.job.classpath.files`, and the URI constructed from `file.makeQualified(conf).toUri()` is stored under the key `mapred.cache.files`. If `file` is not exactly equal to the path portion of the constructed URI, `file` will not be added to the classpath of the task correctly.

Caution The `file` path added must be an absolute path on the JobTracker shared file system, and be only a path. `/user/hadoop/myfile` is correct; `hdfs://host:8020/user/hadoop/myfile` will fail, as will `hadoop/myfile` or `myfile`.

Distributing Archives and Files to Tasks

In addition to items that become available via the classpath, two methods distribute archives and individual files: `DistributedCache.addCacheArchive(URI uri, Configuration conf)` and `DistributedCache.addCacheFile(URI uri, Configuration conf)`. Local file system copies of these items are made on all of the TaskTracker machines, in the work area set aside for this job.

Distributing Archives

The `DistributedCache.addCacheArchive(URI uri, Configuration conf)` method will add an archive to the list of archives to be distributed to the jobs. The URI must have an absolute path and be on the JobTracker shared file system.

If the URI has a fragment, a symbolic link to the archive will be placed in the task working directory as the fragment. The URI `hdfs://host:8020/user/hadoop/myfile#mylink` will result in a symbolic link `mylink` in the task working directory that points to the local file system location that `myfile` was unpacked into at task start. The archive will be unpacked into the local working directory of the task.

The URI will be stored in the configuration under the key `mapred.cache.archives`.

Distributing Files

This `DistributedCache.addCacheFile(URI *uri*, Configuration *conf*)` method will make a copy of the file *`uri`* available to all of the tasks, as a local file system file. The URI must be on the JobTracker shared file system.

If the URI has a fragment, a symbolic link to the URI fragment will be created in the JVM working directory that points to the location on the local file system where the *`uri`* was unpacked into at task start. The directory where `DistributedCache` stores the local copies of the passed items is not the current working directory of the task JVM. This allows the items to be referenced by names that do not have any path components. In particular, executable items may be referenced as `./name`.

To pass a script via the distributed cache, use `DistributedCache.addCacheFile( new URI ("hdfs://host:8020/user/hadoop/myscript.pl"), conf);`. To pass a script so that it may be invoked via `./script`, use `DistributedCache.addCacheFile( new URI("hdfs://host:8020/ user/hadoop/myscript.pl#*script*"), conf);`.

The URI is stored in the configuration key `mapred.cache.files`.

Accessing the DistributedCache Data

Three methods find the locations of the items that were passed to the task via the `DistributedCache` object: `URI JobConf.getResource(name)`, `public static Path[]getLocalCacheArchives (Configuration conf)`, and `public static Path[] getLocalCacheFiles(Configuration conf)`.

Looking Up Names

The `URI JobConf.getResource(name)` method will look up `name` in the classpath. If `name` has a leading slash, this method will search for it in each location in the classpath, and return the URI.

If the job passed a file into `DistributedCache` via the `-files` command or the `DistributedCache.addFileToClassPath(Path file, conf)` method, a `getResource()` call of the file name component, with a leading slash, will return the URI.

Note The standard search rules for resources apply. The cache items will be the last items in the classpath. This does not appear to work for files that are added via `DistributedCache.addFileToClassPath`. The full path is available via the set of paths returned by `DistributedCache.getFileClassPaths()`.

The `DistributedCache.addArchiveToClassPath(jarFileForClassPath, job)` method actually stores the JAR information into the configuration. In the following example, `Utils.setupArchiveFile` builds a JAR file with ten files in it, in the default file system (HDFS in this case). `Utils.makeAbsolute` returns the absolute path.

```
Path jarFileForClassPath = Utils.makeAbsolute(Utils.setupArchiveFile(job, ➥
10, true),job);
DistributedCache.addArchiveToClassPath(jarFileForClassPath, job);
```

Any file that is in the JAR may be accessed via the `getResource()` method of the configuration object. If there were a file `myfile` in the JAR, the call `conf.getResource("/myfile");` would return the URL of the resource. The call `conf. getConfResourceAsInputStream("/myfile");` would return an `InputStream` that, when read, would provide the contents of `myfile` from the JAR.

Looking Up Archives and Files

The `public static Path[]getLocalCacheArchives (Configuration conf)` method returns a list of the archives that were passed via `DistributedCache`. The paths will be in the task local area of the local file system. Any archive passed via the command-line `-libjars` and `-archives` options, or the methods `DistributedCache.addCacheArchive()` and `DistributedCache.addArchiveToClassPath()` and the `JobConf.setJar` line, will have its path returned by this call.

It is possible that the file name portion of your archive will be changed slightly. `DistributedCache` provides the following method to help with this situation:

```
public static String makeRelative(URI cache, Configuration conf)
```

This takes an original archive path and returns the possibly altered file name component.

The `public static Path[] getLocalCacheFiles(Configuration conf)` method returns the set of localized paths for files that are passed via `DistributedCache.addCacheFile` and `DistributedCache.addFileToClassPath` and the command-line option `-files`. The file name portions of the paths may be different from the original file name.

Finding a File or Archive in the Localized Cache

The `DistributedCache` object may change the file name portion of the files and archives it distributes. This is usually not a problem for classpath items, but it may be a problem for non-classpath items. The `Utils.makeRelativeName()` method, described in Table 5-2 provides a way to determine what the file name portion of the passed item was changed to. In addition to the file name portion, the items will be stored in a location relative to the working area for the task on each TaskTracker. Table 5-2 lists the methods provided in the downloadable code that make working with the `DistributedCache` object simpler. These methods are designed to be used in the mapper and reducer methods.

Table 5-2. *Utility Methods Provided in the Examples for Working with the DistributedCache Object*

Method	Description
`Utils.makeRelativeName( name, conf)`	Returns the actual name `DistributedCache` will use for the passed-in `name`.
`Utils.findClassPathArchive( name, conf)`	Returns the actual path on the current machine of the archive `name` that was passed via `DistributedCache.addArchiveToClassPath`.
`Utils.findClassPathFile( name, conf)`	Returns the actual path on the current machine of the file `name` that was passed via `DistributeCacheAddFileToClasspath`.
`Utils.findNonClassPathArchive( name, conf)`	Returns the actual path on the current machine of the archive `name` that was passed via `DistributedCache.addCacheArchive`.
`Utils.findNonClassPathFile( name, conf)`	Returns the actual path on the current machine of the file `name` that was passed via `DistributedCache.addCacheFile`.

Configuring the Hadoop Core Cluster Information

The `JobConf` object provides two basic and critical ways for specifying the default file system: the URI to use for all shared file system paths, and the connection information for the Job-Tracker server. These two items are normally specified in the `conf/hadoop-site.xml` file, but they may be specified on the command line or by setting the values on the `JobConf` object.

Setting the Default File System URI

The default file system URI is normally specified with the `fs.default.name` setting in the `hadoop-site.xml` file, as it is cluster-specific. The value will be `hdfs://NamenodeHostname:PORT`. The `PORT` portion is optional and defaults to `8020`, as of Hadoop 0.18

Note The default value for the file system URI is `file:///`, which stores all files on the local file system. The file system that is used must be a file system that is shared among all of the nodes in the cluster.

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://NamenodeHostname:PORT</value>
 </property>
```

The Hadoop tools, examples, and any application that uses the `GenericOptionParser` class to handle command-line arguments will accept a `-fs hdfs://NamenodeHostname:PORT` command-line argument pair to explicitly set the `fs.default.name` value in the configuration. This will override the value specified in the `hadoop-site.xml` file.

Here's a sample command line for listing files on an explicitly specified HDFS file system:

```
bin/hadoop dfs -fs hdfs://AlternateClusterNamenodeHostname:8020 -ls /
```

You can also use the `JobConf` object to set the default file system:

```
conf.set( "fs.default.name", "hdfs://NamenodeHostname:PORT");
```

Setting the JobTracker Location

The JobTracker location is normally specified with the `mapred.job.tracker` setting in the `hadoop-site.xml` file, as it is cluster-specific. The value will be `JobTrackerHostname:PORT`. Through Hadoop 0.19, there is not a standard for the `PORT`. Many installations use a port one higher that the HDFS port.

■ **Note** The default value for the JobTracker location is `local`, which will result in the job being executed by the JVM that submits it. The value `local` is ideal for testing and debugging new MapReduce jobs. It is important to ensure that any required Hadoop configuration files are in the classpath of the test jobs.

```
<property>
  <name>mapred.job.tracker</name>
  <value>JobtrackerHostname:PORT</value>
 </property>
```

Here's a sample command line explicitly setting the JobTracker for job control for listing jobs:

```
bin/hadoop job -jt AlternateClusterJobtrackerHostname:8021 -list
```

And here's how to use the `JobConf` object to set the JobTracker information:

```
conf.set( "mapred.job.tracker", "JobtrackerHostname:PORT");
```

The Mapper Dissected

All Hadoop jobs start with a mapper. The reducer is optional. The class providing the map function must implement the `org.apache.hadoop.mapred.Mapper` interface, which in turn requires the interfaces `org.apache.hadoop.mapred.JobConfigurable` and `org.apache.hadoop.io.Closeable`. The Hadoop framework provides `org.apache.hadoop.mapred.MapReduceBase` from which to derive mapper and reducer classes. The `JobConfigurable` and `Closable` implementations are empty methods. In the utilities supplied with this book's downloadable code is `com.apress.hadoopbook.utils.MapReduceBase`, which provides more useful implementations.

Note The interface `org.apache.hadoop.io.Closeable` will be replaced with `java.io.Closeable` in a later release.

This section examines the sample mapper class `SampleMapperRunner.java`, which is available with the rest of the downloadable code for this book. When run as a Java application, this example accepts all of the standard Hadoop arguments and may be run with custom bean context and definitions:

```
bin/hadoop jar hadoopprobook.jar ➥
com.apress.hadoopbook.examples.ch5.SampleMapperRunner -D ➥
mapper.bean.context=mycontext.xml -D mapper.bean.name=mybean -files ➥
mycontext.xml -deleteOutput
```

where:

- `bin/hadoop jar` is the standard Hadoop program invocation.
- `hadoopprobook.jar com.apress.hadoopbook.examples.ch5.SampleMapperRunner` specifies the JAR file to use and the main class to run.
- `-D mapper.bean.context=mycontext.xml` and `-D mapper.bean.name=mybean` specify that the string `mycontext.xml` is stored in the configuration under the key `mapper.bean.context`, and that the string `mybean` is stored in the configuration under the key `mapper.bean.name`.
- `-files mycontext.xml` causes the file `mycontext.xml` to be copied into HDFS, and then unpacked and made available in the working directory of each task run by the job. The working directory is in the task classpath. `mycontext.xml` may have a directory path component, and not be just a stand-alone file name. The path and file name provided must be a path that can be opened from the current working directory.

Note If you are using the value `local` as the value of the `mapred.task.tracker` configuration key, using the `DistributedCache` object is less effective, as the task cannot change working directories.

- `--deleteOutput`, which must be the last argument, causes the output directory to be deleted before the job is started. This is convenient when running the job multiple times.

Mapper Methods

For the mapper, the framework will call three methods:

- `configure()` method, defined in the `Configurable` interface
- `map()` method, defined in the `Mapper` interface
- `close()` method, defined in the `Closable` interface

The following sections discuss these methods in detail.

■**Note** The framework uses the static method `org.apache.hadoop.util.ReflectionUtils.<T>newInstance(Class<T> theClass, Configuration conf)` to create instances of objects that need a copy of the configuration. This will create the instance using the no-argument constructor. If the class is an instance of `Configurable`, `newInstance` will call the `setConf` method with the supplied configuration. If the class is an instance of `JobConfiguration`, `newInstance` will call the `configure` method. Any exceptions that are thrown during the construction or initialization of the instance are rethrown as `RuntimeExceptions`.

The configure() Method

The `void JobConfigurable.configure(JobConf job)` method, defined in `org.apache.hadoop.conf.Configurable`, is called exactly one time per map task as part of the initialization of the `Mapper` instance. If an exception is thrown, this task will fail. The framework may attempt to retry this task on another host if the allowable number of failures for the task has not been exceeded. The methods `JobConf.getMaxMapAttempts()` and `JobConf.setMaxMapAttempts(int n)` control the number of times a map task will be retried if the task fails. The default is four times.

It is considered good practice for any `Mapper` implementation to declare a member variable that the `configure()` method uses to store a reference to the passed-in `JobConf` object. The `configure()` method is also used for loading any Spring application context or initializing resources that are passed via `DistributedCache`.

Listing 5-1 shows the `configure()` method used in `SampleMapperRunner.java` (the example available with the downloadable code for this chapter).

Listing 5-1. *configure Method from SampleMapperRunner.java*

```
/** Sample Configure method for a map/reduce class.
 * This method assumes the class derives from {@link MapReduceBase}
 * and saves a copy of the JobConf object, the taskName
 * and the taskId into member variables.
 * and makes an instance of the output key and output value
 * objects as member variables for the
 * map or reduce to use.
 *
 * If this method fails the Tasktracker will abort this task.
 * @param job The Localized JobConf object for this task
 */
public void configure(JobConf job) {
    super.configure(job);
    LOG.info("Map Task Configure");
    this.conf = job;
    try {
        taskName = conf.getJobName();
        taskId = TaskAttemptID.forName(conf.get("mapred.task.id"));
        if (taskName == null || taskName.length() == 0) {
            /** if the job name is essentially unset make something up. */
            taskName = taskId.isMap() ? "map." : "reduce."
                    + this.getClass().getName();
        }

        /**
         * These casts are safe as they are checked by the framework
         * earlier in the process.
         */
        outputKey = (K2) conf.getMapOutputKeyClass().newInstance();
        outputValue = (V2) conf.getMapOutputValueClass().newInstance();
    } catch (RuntimeException e) {
        LOG.error("Map Task Failed to initialize", e);
        throw e;
    } catch (InstantiationException e) {
        LOG.error(
                "Failed to instantiate the key or output value class",
                e);
        throw new RuntimeException(e);
    } catch (IllegalAccessException e) {
        LOG
                .error(
 "Failed to run no argument constructor for key or output value objects",
                        e);
        throw new RuntimeException(e);
    }
    LOG.info(taskId.isMap() ? "Map" : "Reduce" + " Task Configure complete");

}
```

In this example, `K2` is the map output key type, which defaults to the reduce output key type, which defaults to `LongWritable`. `V2` is the map output value key type, which defaults to the reduce output value type, which defaults to `Text`.

This `configure()` method saves a copy of the `JobConf` object `taskId` and `taskName` into member variables. This method also instantiates a local instance of the key and value classes, to be used during the map() method calls. By using the isMap method on the taskId, you can take different actions for map and reduce tasks in the configure() and close() methods. This becomes very useful when a single class provides both a map method and a reduce method.

The map( ) Method

A call to the void map(K1 key, V1 value, OutputCollector<K2,V2> output, Reporter reporter) throws IOException method, defined in org.apache.hadoop.mapred.Mapper, will be made for every record in the job input. No calls will be made to the map() method in an instance before the configure() method completes.

If the job is configured for running multithreaded map tasks, as follows, there may be multiple simultaneous calls to the map() method.

```
jobConf.setMapRunnerClass(MultithreadedMapRunner.class);
jobConf.setInt("mapred.map.multithreadedrunner.threads", 10);
```

When running multithreaded, each map() call will have a different key and value object. The output and reporter objects are shared. The default number of threads for a multithreaded map task is ten.

The contents of the key object and the contents of the value object are valid only during the map() method call. The framework will reset the object contents with the next key/value pair prior to the next call to map().

The class converting the input into records is responsible for defining the types of K1 and V1. The standard textual input format, KeyValueTextInput, defines K1 and V1 to be of type Text.

K2 and V2 are defined by the JobConf.setMapOutputKeyClass(clazz) and JobConf.setMapOutputValueClass(clazz) methods. The types of K2 and V2 default to the classes set for the reduce key and value output classes. The reduce key and value output classes are set by JobConf.setOutputKeyClass(clazz) and JobConf.setOutputValueClass(clazz). The defaults for K2 and V2 are LongWritable and Text, respectively. You can explicitly configure the map output key and value classes, as follows:

```
jobConf.setMapOutputKeyClass(MyMapOutputKey.class);
jobConf.setMapOutputValueClass(MyMapOutputValue.class)
```

If a map output class is set, the corresponding reduce input class is also set to the class. If the map output key class is changed to BytesWritable, the Reducer.reduce's key type will be BytesWritable.

The close() Method

The void close() method, defined in java.io.Closable, is called one time after the last call to the map() method is made by the framework. This method is the place to close any open files or perform any status checking. Unless your configure() method has saved a copy of the JobConf object, there is little interaction that can be done with the framework. The close() method example in Listing 5-2 checks the task status based on the ratio of exceptions to input keys.

Listing 5-2. *close Method from SampleMapperRunner.java*

```
/** Sample close method that sets the task status based on how
 * many map exceptions there were.
 * This assumes that the reporter object passed into the map method was saved and
 *  that the JobConf object passed into the configure method was saved.
 */
public void close() throws IOException {
    super.close();
    LOG.info("Map task close");
    if (reporter != null) {
        /**
         * If we have a reporter we can perform simple checks on the
         * completion status and set a status message for this task.
         */
        Counter mapExceptionCounter = reporter.getCounter(taskName,
                "Total Map Failures");
        Counter mapTotalKeys = reporter.getCounter(taskName,
                "Total Map Keys");
        if (mapExceptionCounter.getCounter() == mapTotalKeys
                .getCounter()) {
            reporter.setStatus("Total Failure");
        } else if (mapExceptionCounter.getCounter() != 0) {
            reporter.setStatus("Partial Success");
        } else {
            /** Use the Spring set bean to show we did get the values. */
 reporter.incrCounter( taskName, getSpringSetString(), getSpringSetInt());
            reporter.setStatus("Complete Success");
        }
    }
    /**
     * Ensure any HDFS files are closed here, to force them to be
     * committed to HDFS.
     */
}
```

The `close()` method in Listing 5-2 will report success or failure status back to the framework, based on an examination of the job counters. It assumes that the `map()` method reported an exception under the counter, `Total Map Failure`, in the counter group `taskName`, and the number of keys received is in the counter, `Total Map Keys`, in the counter group `taskName`.

If there are no exceptions, the method will report the task status as “Complete Success.” If there are some exceptions, the status is set to “Partial Success,” If the exception count equals the key count, the status is set to “Total Failure.”

This example also logs to counters with the values received from the Spring initialization. I found the Spring value-based counters useful while working out how to initialize map class member variables via the Spring Framework, as described after the discussion of the mapper class declaration and member fields.

Mapper Class Declaration and Member Fields

It is a best practice to capture the `JobConf` object passed in the `configure()` method into a member variable. It is also a good practice to instantiate member variables, or thread local variables, for any key or value that would otherwise be created in the body of the `map()` method. Having the `TaskAttemptId` available is also useful, as it is easy to determine if this is the map phase or the reduce phase of a job.

It is convenient to capture the output collector and the reporter into member fields so that they may be used in the `close()` method. This has a downside in that they can be captured only in the `map()` method, requiring extra code in that inner loop.

Listing 5-3 shows an example that declares a number of local variables, which are initialized by the `configure()` method for use by the `map()` and `close()` methods.

Listing 5-3. *Class and Member Variable Declarations from SampleMapperRunner.java*

```
/**
 * Sample Mapper shell showing various practices
 *
 * K1 and V1 will be defined by the InputFormat. K2 and V2 will be the
 * {@link JobConf#getOutputKeyClass()} and
 * {@link JobConf#getOutputValueClass()}, which by default are LongWritable
 * and Text. K1 and V1 may be explicitly set via
 * {@link JobConf#setMapOutputKeyClass(Class)} and
 * {@link JobConf#setMapOutputValueClass(Class)}. If K1 and V1 are
 * explicitly set, they become the K1 and V1 for the Reducer.
 *
 * @author Jason
 *
 */
public static class SampleMapper<K1, V1, K2, V2> extends MapReduceBase
        implements Mapper<K1, V1, K2, V2> {

    /**
     * Create a logging object or you will never know what happened in your
     * task.
     */

    /** Used in metrics reporting. */
    String taskName = null;
    /**
     * Always save one of these away. They are so handy for almost any
     * interaction with the framework.
     */
    JobConf conf = null;
    /**
     * These are nice to save, but require a test or a set each pass through
     * the map method.
     */
    Reporter reporter = null;
    /** Take this early, it is handy to have. */
    TaskAttemptID taskId = null;

    /**
     * If we are constructing new keys or values for the output, it is a
     * best practice to generate the key and value object once, and reset
     * them each time. Remember that the map method is an inner loop that
     * may be called millions of times. These really can't be used without
     * knowing an actual type
     */
    K2 outputKey = null;
    V2 outputValue = null;
```

Initializing the Mapper with Spring

Many installations use the Spring Framework to manage the services employed by their applications. One of the more interesting issues is how to use Spring in environments where Spring does not have full control over the creation of class instances. Spring likes to be in full control of the application and manage the creation of all of the Spring bean objects. In the Hadoop case, the Hadoop framework is in charge and will create the object instances. The examples in this section demonstrate how to use Spring to initialize member variables in the mapper class. The same techniques apply to the reducer class.

Listing 5-4 shows the bean file used in the Spring example. The file `mapper.bean.context.xml` in the downloadable examples `src/config` directory is the actual file used.

Listing 5-4. *Simple Bean Resource File for the Spring-Initialized Task*

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans ➥
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context ➥
        http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="SampleMapperJob.mapper.bean.name"
       class="com.apress.hadoopbook.examples.ch5.SampleMapperRunner.SampleMapper"
       lazy-init="true"
       scope="singleton">
       <description> Simple bean definition to provide an example for
       using Spring to initialize context in a Mapper class.</description>
       <property name="springSetString"><value>SetFromDefaultFile</value></property>
       <property name="springSetInt"><value>37</value></property>
  </bean>
</beans>
```

Creating the Spring Application Context

To create an application context, you need to provide Spring with a resource set from which to load bean definitions. Being very `JobConf`-oriented, I prefer to pass the names of these resources, and possibly the resources themselves, to my tasks via the `JobConf` and `DistributedCache` objects.

The example in Listing 5-5 extracts the set of resource names from the `JobConf` object, and if not found, will supply a default set of resource names. This follows the Hadoop style of using comma-separated elements to store multiple elements in the configuration. The set of resources names are unpacked and passed to the Spring Framework. Each of these resources must be in the classpath, which includes the task working directory.

At the very simplest, the user may specify the specific Spring configuration files on the command line via the `-files` argument, when the `GenericOptionsParser` is in use. The mapper class will need to determine the name of the file passed in via the command line. For the example, set up the Spring initialization parameters on the application command line as follows:

```
hadoop jar appJar main-class -files spring1.xml,spring2,xml,spring3.xml ➥
-D mapper.bean.context=spring1.xml
```

■**Note** In the command-line specification, the `-D mapper.bean.context=value` argument must come after the main class reference to be stored in the job configuration. If it comes before the `jar` argument, it will become a Java system property.

The example in Listing 5-5 copies `spring1.xml`, `spring2.xml`, and `spring3.xml` from the local file system into HDFS, and then copies them to the task local directory and creates symbolic links from the local copy to the task working directory. The configuration parameter `mapper.bean.context` tells the map task which bean file to load. In the example, `SampleMapperRunner` looks up the configuration entry `mapper.bean.context` to determine which bean files to use when creating the application context.

Listing 5-5. *Extracting the Resource File Names from the JobConf Object and Initializing the Spring Application Context (from utils.Utils.java)*

```
/**
 * Initialize the Spring environment. This is of course completely
 * optional.
 *
 * This method picks up the application context from a file, that is in
 * the classpath. If the file items are passed through the
 * {@link DistributedCache} and symlinked
 * they will be in the classpath.
 *
 * @param conf The JobConf object to look for the Spring config file names.
 * If this is null, the default value is used.
 * @param contextConfigName The token to look under in the config for the names
 * @param defaultConfigString A default value
 * @return TODO
 *
 */
public static ApplicationContext initSpring(JobConf conf, String contextConfigName,
        String defaultConfigString) {
    /**
     * If you are a Spring user, you would initialize your application
     * context here.
     */
    /** Look up the context config files in the JobConf, provide a default value. */
    String applicationContextFileNameSet =
        conf == null ? defaultConfigString :
            conf.get( contextConfigName, defaultConfigString);
    LOG.info("Map Application Context File "
            + applicationContextFileNameSet);

    /** If no config information was found, bail out. */
    if (applicationContextFileNameSet==null) {
        LOG.error( "Unable to initialize Spring configuration using "
         + applicationContextFileNameSet );
        return null;
    }
    /** Attempt to split it into components using the config
      * standard method of comma separators. */
    String[] components = StringUtils.split(applicationContextFileNameSet, ",");

    /** Load the configuration. */
    ApplicationContext applicationContext =
        new ClassPathXmlApplicationContext( components);

    return applicationContext;
}
```

Using Spring to Autowire the Mapper Class

Once the Spring application context has been created, the task may instantiate beans. The confusing issue is that the mapper class has already been instantiated, so how can Spring be forced to initialize/autowire that class?

Accomplishing this autowiring requires two things. The first is that the bean definition to be used must specify lazy initialization, to prevent Spring from creating an instance of the bean when the application context is created. The second is to know the bean name/ID of the mapper class.

The example in Listing 5-6 makes some assumptions about how application contexts and task beans are named, and can be easily modified for your application.

Listing 5-6. *Example of a Spring Task Initialization Method*

```
/** Handle Spring configuration for the mapper.
 * The bean definition has to be <code>lazy-init="true"</code>
 * as this object must be initialized.
 * This will fail if Spring weaves a wrapper class for AOP around
 * the configure bean.
 *
 * The bean name is extracted from the configuration as
 * mapper.bean.name or reducer.bean.name
 * or defaults to taskName.XXXX.bean.name
 *
 * The application context is loaded from mapper.bean.context
 * or reducer.bean.context and may be a set of files
 * The default is jobName.XXX.bean.context
 *
 * @param job The JobConf object to look up application context ➥
    files and bean names in
 * @param RuntimeException if the application context can not be ➥
    loaded or the initializtion requires delegation of the task object.
 */
 void springAutoWire(JobConf job) {
    String springBaseName = taskId.isMap()? "mapper.bean": "reducer.bean";

    /** Construct a bean name for this class using the configuration
      * or a default name. */
    String beanName = conf.get(springBaseName + ".name",
        taskName + "." + springBaseName + ".name" );
    LOG.info("Bean name is " + beanName);
    applicationContext = Utils.initSpring(job, springBaseName
     + ".context", springBaseName + ".context.xml");
    if (applicationContext==null) {
        throw new RuntimeException(
 "Unable to initialize spring configuration for " + springBaseName);
    }
    AutowireCapableBeanFactory autowire =
  applicationContext.getAutowireCapableBeanFactory();
    Object mayBeWrapped = autowire.configureBean( this, beanName);
    if (mayBeWrapped != this) {
        throw new RuntimeException( "Spring wrapped our class for " + beanName);
    }
}
```

In Listing 5-6, a base name is constructed for looking up information in the configuration via the following:

```
String springBaseName = taskId.isMap()? "mapper.bean": "reducer.bean";
```

The example builds a context file name key, which will be `mapper.bean.context` in the case of a map, to look up the application context information in the configuration. If a value is found, it is treated as a comma-separated list of bean resource files to load. The application context is loaded and saved in the member variable `applicationContext`:

```
applicationContext = SpringUtils.initSpring(job, springBaseName
   + ".context", springBaseName + ".context.xml");
```

A default bean file is used if no value is found. In this example, the file is `mapper.bean.context.xml`.

A bean name key `mapper.bean.name`, with a default value of `mapper.bean.name`, is looked up in the configuration. This is the bean that will be used to configure the task. The following line constructs the bean name to use:

```
String beanName = conf.get(springBaseName + ".name", taskName + "."
   + springBaseName + ".name" );
```

An autowire-capable bean factory is extracted from the application context via the following:

```
AutowireCapableBeanFactory autowire =
  applicationContext.getAutowireCapableBeanFactory();
```

The following line actually causes Spring to initialize the task:

```
Object mayBeWrapped = autowire.configureBean( this, beanName);
```

The code must ensure that Spring did not return a delegator object when it was initializing the task from the bean definition:

```
if (mayBeWrapped != this) {
    throw new RuntimeException( "Spring wrapped our class for " + beanName);
}
```

■ **Note** This example does not handle the case where Spring returns a delegator object for the task. To handle this case, the `map()` method would need to be redirected through the delegated object.

Partitioners Dissected

A core part of the MapReduce concept requires that map outputs be split into multiple streams called *partitions*, and that each of these partitions is fed to a single reduce task. The reduce contract specifies that each reduce task will be given as input the fully sorted set of keys and their values in a particular partition. The entire partition is the input of the reduce task. For the framework to satisfy this contract, a number of things have to happen first. The outputs of each map task are partitioned and sorted. The partitioner is run in the context of the map task.

The Hadoop framework provides several partitioning classes and a mechanism to specify a class to use for partitioning. The actual class to be used must implement the `org.apache.hadoop.mapred.Partitioner` interface, as shown in Listing 5-7. The piece that provides a partition number is the `getPartition()` method:

```
int getPartition(K2 key, V2 value, int numPartitions)
```

Note that both the key and the value are available in making the partition choice.

Listing 5-7. *The Partitioner Interface in Hadoop 0.19.0*

```
/**
 * Partitions the key space.
 *
 * <p><code>Partitioner</code> controls the partitioning of the keys of the
 * intermediate map-outputs. The key (or a subset of the key) is used to derive
 * the partition, typically by a hash function. The total number of partitions
 * is the same as the number of reduce tasks for the job. Hence this controls
 * which of the <code>m</code> reduce tasks the intermediate key (and hence the
 * record) is sent for reduction.</p>
 *
 * @see Reducer
 */
public interface Partitioner<K2, V2> extends JobConfigurable {

  /**
   * Get the partition number for a given key (hence record) given the total
   * number of partitions i.e. number of reduce tasks for the job.
   *
   * <p>Typically a hash function on a all or a subset of the key.</p>
   *
   * @param key the key to be paritioned.
   * @param value the entry value.
   * @param numPartitions the total number of partitions.
   * @return the partition number for the <code>key</code>.
   */
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

The key and value will be streamed into the partition number that this function returns. Each key/value pair output by the `map()` method has the partition number determined and is then written to that map local partition. Each of these map local partition files is sorted in key order by the class returned by the `JobConf.getOutputKeyComparator()` method.

For each reduce task, the framework will collect all the reduce task's partition pieces from each of the map tasks and merge-sort those pieces. The results of the merge-sort are then fed to the `reduce()` method. The merge-sort is also done by the class returned by the `JobConf.getOutputKeyComparator()` method.

The output of a reduce task will be written to the `part-`*XXXXX* file, where the *XXXXX* corresponds to the partition number.

The Hadoop framework provides the following partitioner classes:

- `HashPartitioner`, which is the default
- `TotalOrderPartitioner`, which provides a way to partition by range
- `KeyFieldBasedPartitioner`, which provides a way to partition by parts of the key

The following sections describe each of these partitioners.

The HashPartitioner Class

The default partitioner, `org.apache.hadoop.mapred.lib.HashPartitioner`, simply uses the hash code value of the key as the determining factor for partitioning. Listing 5-8 shows the actual code from the default partitioner used by Hadoop. The partition number is simply the hash value of the key modulus the number of partitions.

Listing 5-8. *The HashCode Partitioner from Hadoop 0.19.0*

```
/** Partition keys by their {@link Object#hashCode()}. */
public class HashPartitioner<K2, V2> implements Partitioner<K2, V2> {

  public void configure(JobConf job) {}

  /** Use {@link Object#hashCode()} to partition. */
  public int getPartition(K2 key, V2 value,
                          int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }

}
```

The hash value is converted to a positive value, `(key.hashCode() & Integer.MAX_VALUE)`, to ensure that the partition will be a positive integer. The resulting number has modulus the number of reduce tasks applied, `% numReduceTasks`, and the result returned. This produces a positive number between 0 and one less than the number of partitions.

The TotalOrderPartitioner Class

The `TotalOrderPartitioner`, `org.apache.hadoop.mapred.lib.TotalOrderPartitioner`, relies on a file that provides the class with range information. With this information, the partitioner is able to determine which range a key/value pair belongs in and route it to the relevant partition.

Note The `TotalOrderParitioner` grew out of the `TetraSort` example package. Jim Gray introduced a contest called the TeraByteSort, which was a benchmark to sort one terabyte of data and write the results to disk. In 2008, Yahoo! produced a Hadoop version of the test that completed in 209 seconds (`http://developer.yahoo.net/blogs/hadoop/2008/07/apache_hadoop_wins_terabyte_sort_benchmark.html`). The code is included with the Hadoop examples as `bin/hadoop jar hadoop-*-examples.jar terasort in-dir out-dir`. The class file is `org.apache.hadoop.examples.terasort.TeraSort`.

Building a Range Table

The `org.apache.hadoop.mapred.lib.InputSampler` class is used to generate a range partitioning file for arbitrary input sets. This class will sample the input to build an approximate range table.

This sampling strategy will take no more than the specified number of samples total from the input. The user may specify a maximum number of input splits to look in as well. The actual number of records read from each input split varies based on the number of splits and the number of records in the input split.

The Hadoop framework controls how the input is split based on the number of input files, the input format, the input file size, and the minimum split size and the HDFS block size. Let's look at a few examples of running `InputSampler` from the command line.

In the following example, the argument set `-splitSample 1000 10` will sample a total of 1,000 input records out of no more than 10 input splits.

```
bin/hadoop jar hadoop-0.19.0-core.jar org.apache.hadoop.mapred.lib.InputSampler ➥
-inFormat org.apache.hadoop.mapred.KeyValueTextInputFormat ➥
-keyClass org.apache.hadoop.io.Text -r 15 -splitSample 1000 10 csvin csvout
```

If there are 10 or more input splits, each of which has more than 100 records, the first 100 records from each input split will be used for samples. The input is loaded from the directory `csvin`, and is parsed by the `KeyValueTextInputFormat` class. The range file is written to `csvout`, and the argument set `-r 15` sets up the output for a job with 15 output partitions. The input splits are examined in the order in which they are returned by `InputFormat`.

The next example takes 1,000 samples from roughly 10 input splits. The input splits are sampled in a random order, and the records from each split read are sequentially.

```
bin/hadoop jar hadoop-0.19.0-core.jar org.apache.hadoop.mapred.lib.InputSampler ➥
-inFormat org.apache.hadoop.mapred.KeyValueTextInputFormat ➥
-keyClass org.apache.hadoop.io.Text -r 15 -splitRandom .1 1000 10 csvin csvout
```

Each record has a 0.1% chance of being selected. The `-splitRandom .1 1000 10` argument set specifies the percentage, the total samples, and the maximum splits to sample. If the 1,000 samples are not selected after processing the recommended number of splits, more splits will be sampled. The index is set up for 15 reduce tasks, and the input comes from `csvin`. The index is written to `csvout`. The splits to examine are selected randomly.

In the final example, the argument set `-splitInterval .01 10` will examine no more than 10 input splits and take one record in 100 from each split.

```
bin/hadoop jar hadoop-0.19.0-core.jar org.apache.hadoop.mapred.lib.InputSampler ➥
-inFormat org.apache.hadoop.mapred.KeyValueTextInputFormat ➥
-keyClass org.apache.hadoop.io.Text -r 15 -splitInterval .01 10 csvin csvout
```

The frequency parameter defines how many records will be sampled. For a frequency of `0.1`, as in this example, one record in 10 will be used. For a frequency of `0.01`, one record in 100 will be used. The index is set up for 15 reduce tasks. The input comes from `csvin`, and the index is written to `csvout`.

Using the TotalOrderPartitioner

Once an index is generated, a job may be set up to use the `TotalOrderPartitioner` and the index. Three configuration settings are required for this to work:

- The partitioner must be set to `TotalOrderPartitioner` in the `JobConf` object via `conf.setPartitionerClass(TotalOrderPartitioner)`.
- The partitioning index must be specified via the configuration key `total.order.partitioner.path`:

  ```
  conf.set("total.order.partitioner.path", "csvout");
  ```

- The sort type for the keys must also be specified. If the binary representation of the keys is the correct sorting, the Boolean field `total.order.partitioner.natural.order` should be set to `true` in the configuration. If the binary representation of the keys is not the correct sort, the Boolean field `total.order.partitioner.natural.order` must be set to `false`. This Boolean field is set as follows:

  ```
  conf.setBoolean("total.order.partitioner.natural.order");
  ```

  If the binary representation of the key is the correct sort order, a binary trie (an ordered tree structure; see `http://en.wikipedia.org/wiki/Trie`) will be constructed and used for searching; otherwise, a binary search based on the output key comparator will be used.

Here's an example of how to put all this together:

```
TotalOrderPartitioner.setPartitionFile(conf,"csvout");
conf.setPartitionerClass(TotalOrderPartitioner.class);
conf.set("total.order.partitioner.natural.order",false);
conf.setNumReduceTasks (15);
```

In this example, `csvin` is the input file, and `csvout` is the index file. The `csvout` file was set up for 15 reduce tasks, and requires the comparator rather than binary comparison.

The KeyFieldBasedPartitioner Class

The `KeyFieldBasedPartitioner`, `org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner`, provides the job with a way of using only parts of the key for comparison purposes. The primary concept is that the keys may be split into pieces based on a piece separator string. Each piece is then numbered from 1 to *N*, and each character of each piece numbered from 1 to *M*.

The separator string is defined by the configuration key `map.output.key.field.separator` and defaults to the tab character. It may be set to another string, `str`, as follows:

```
conf.set(map.output.key.field.separator, str);
```

This is functionally equivalent to using the `String.split(Pattern.quote(str))` call on each key and treating the resulting array as if indexes were one-based instead of zero-based.

If the separator is `X` and the key is `oneXtwoXthree`, the pieces will be `1) one`, `2) two`, `3) three`.

Referencing individual characters within the pieces is also one-based rather than zero-based, with the index 0 being the index of the last character of the key part. For the first key piece in the preceding example, the string `one`, the characters will be `1) o`, `2) n`, `3) e`, `0) e`. Note that both `3` and `0` refer to `e`, which is the last character of the key piece.

■**Note** In addition to the one-based ordinal position within the key piece, the last character of the key piece may also be referenced by `0`.

The key pieces to compare are specified by setting the key field partition option, via the following:

```
conf. setKeyFieldPartitionerOptions(str).
```

The `str` format is very similar to the key field-based comparator.

The Javadoc from Hadoop 0.19.0 for `KeyFieldBasedPartitioner` provides the following definition:

> *Defines a way to partition keys based on certain key fields (also see KeyFieldBasedComparator). The key specification supported is of the form -k pos1[,pos2], where, pos is of the form f[.c][opts], where f is the number of the key field to use, and c is the number of the first character from the beginning of the field. Fields and character posns are numbered starting with 1; a character position of zero in pos2 indicates the field's last character. If '.c' is omitted from pos1, it defaults to 1 (the beginning of the field); if omitted from pos2, it defaults to 0 (the end of the field).*

In plain English, `-k#` selects piece # for the comparison, and `-k#1,#2` selects the pieces from `#1` through `#2`. In the preceding example, `-k1` selects `oneX` as the portion of the key to use for comparison, and `-k1,1` selects `one` as the portion of the key to use for comparison.

There is also the facility to select a start and stop point within an individual key. The option `-k1.2,1` is equivalent to `-k1.2,1.0`, and selects `ne` from the `one` for comparison.

You may also span key pieces. `-k1.2,3.2` selects `eXtwoXth` as the comparison region from the sample key. It means to start with key piece 1, character 2 and end with key piece 3 character 2.

Note If your key specification may touch the last key piece, it is important to terminate with the last character of the key. Otherwise, the current code (as of Hadoop 0.19.0) will generate an `ArrayIndexOutOfBoundsException` as it tries to use the missing separator string. In this section's example, `-k3,3` would work, but `-k3` would throw the exception.

The Reducer Dissected

The reducer has a very similar shape to the mapper. The class may provide `configure()` and `close()` methods. All of the mapper good practices of saving the `JobConf` object and making instances of the output key and output value objects apply to the reducer as well.

The key difference is in the `reduce()` method. Unlike the `map()` method, which is given a single key/value pair on each invocation, each `reduce()` method invocation is given a key and all of the values that share that key.

The reducer is an operator on groups. The default is to define a group as all values that share a key. Common uses for reduce tasks are to suppress duplicates in datasets or to segregate ranges and order output of large datasets.

In the example shown in Listing 5-9, notice that the signature of the `reduce()` method contains an `Iterator<V>`, an iterator over the values that share `key`. The identity reducer simply outputs each value in the iterator.

Listing 5-9. *The Identity Reducer from Hadoop Core 0.19.0*

```
/** Performs no reduction, writing all input values directly to the output. */
public class IdentityReducer<K, V>
    extends MapReduceBase implements Reducer<K, V, K, V> {

  /** Writes all keys and values directly to output. */
  public void reduce(K1 key, Iterator<V1> values,
                     OutputCollector<K2, V2> output, Reporter reporter)
    throws IOException {
    while (values.hasNext()) {
      output.collect(key, values.next());
    }
  }

}
```

The `configure()` and `close()` methods have the same requirements and suggested usage as the corresponding mapper methods.

It is generally recommended that you do not make a copy of all of the value objects, as there may be very many of these objects.

Note In one of my early applications, I assumed that there would never be more than a small number of values per key. The reduce tasks started experiencing out-of-memory exceptions. It turned out that there were often more than 150,000 values per key!

It is possible to simulate a secondary sort/grouping of the values by setting the output value grouping. To do this requires the cooperation of the `OutputComparator`, `OutputPartitioner`, and `OutputValueGroupingComparator`. See this book's appendix for more information.

By default, the input key and value types are the same as the output key and value types, and are set by the `conf.setOutputKeyClass(class)` and `conf.setOutputValueClass(class)` methods. The defaults are `LongWritable` and `Text`, respectively.

If the map output keys must be different, using `conf.setMapOutputKeyClass(class)` and `conf.setMapOutputValueClass(class)` will also change the expected input key and value for the reduce task.

A Simple Transforming Reducer

Listing 5-10 shows the simple transformational reducer, `SimpleReduceTransformingReducer.java`, used in this chapter's `SimpleReduce.java` example.

Listing 5-10. *Transformational Reducer in SimpleReduceTransformingReducer.java*

```
/** Demonstrate some aggregation in the reducer
 *
 * Produce output records that are the key, the average, the count,
 * the min, max and diff
 *
 * @author Jason
 *
 */
public class SimpleReduceTransformingReducer extends MapReduceBase implements
        Reducer<LongWritable, LongWritable, Text, Text> {

    /** Save object churn. */
    Text outputKey = new Text();
    Text outputValue = new Text();

    /** Used in building the textual representation of the output key and values. */
    StringBuilder sb = new StringBuilder();
    Formatter fmt = new Formatter(sb);
```

```
@Override
public void reduce(LongWritable key, Iterator<LongWritable> values,
        OutputCollector<Text, Text> output,
        Reporter reporter) throws IOException {
    /** This is a bad practice, the transformation of
      * the key should be done in the map. */
    reporter.incrCounter("Reduce Input Keys", "Total", 1);
    try {
        long total = 0;
        long count = 0;
        long min = Long.MAX_VALUE;
        long max = 0;

        /** Examine each of the values that grouped with this key. */
        while (values.hasNext()) {
            final long value = values.next().get();
            if (value>max) {
                max = value;
            }
            if (value<min) {
                min = value;
            }
            total += value;
            count++;
        }

        sb.setLength(0);
        fmt.format("%12d %3d %12d %12d %12d", total/count,
                   count, min, max, max-min);
        fmt.flush();
        outputValue.set(sb.toString());

        sb.setLength(0);
        fmt.format("%4d", key.get());
        outputKey.set(sb.toString());

        reporter.incrCounter("Reduce Output Keys", "Total", 1);
        output.collect(outputKey, outputValue);

    } catch( Throwable e) {
        reporter.incrCounter("Reduce Input Keys", "Exception", 1);
        if (e instanceof IOException) {
            throw (IOException) e;
        }
```

```
            if (e instanceof RuntimeException) {
                throw (RuntimeException) e;
            }
            throw new IOException(e);
        }

    }
}
```

It begins by establishing several member variables that will be used in the `reduce()` method to save object generation:

```
/** Save object churn. */
    Text outputKey = new Text();
    Text outputValue = new Text();

    /** Used in building the textual representation of the output key and values. */
    StringBuilder sb = new StringBuilder();
    Formatter fmt = new Formatter(sb);
```

The working body of the `reduce()` method is within a `try` block that catches `Throwables`, and the input count, output count, and failure count are reported to the framework:

```
reporter.incrCounter("Reduce Input Keys", "Total", 1);
try {
    ....
    reporter.incrCounter("Reduce Output Keys", "Total", 1);
    output.collect(outputKey, outputValue);

} catch( Throwable e) {
    reporter.incrCounter("Reduce Input Keys", "Exception", 1);
```

In the body of the example in Listing 5-10, each value that is passed in is examined and aggregated:

```
/** Examine each of the values that grouped with this key. */
while (values.hasNext()) {
    final long value = values.next().get();
    if (value>max) {
        max = value;
    }
    if (value<min) {
        min = value;
    }
    total += value;
    count++;
}
```

Finally, the output key and value are constructed with the aggregated data:

```
sb.setLength(0);
fmt.format("%12d %3d %12d %12d %12d", total/count, count, min, max, max-min);
fmt.flush();
outputValue.set(sb.toString());

sb.setLength(0);
fmt.format("%4d", key.get());
outputKey.set(sb.toString());
```

The example that runs this reducer also uses an output grouping comparator that groups the records in sets of ten. The comparator `Utils.GroupByLongGroupingComparator.java` (supplied with the downloadable code for this chapter) handles grouping `LongWritable` values in sets of 10, 0–9, 10–19, and so on.

The following is the core code in `SimpleReduce` that sets up the job that runs `SimpleReduceTransformingReducer`:

```
job.setInputFormat(KeyValueTextInputFormat.class);
FileInputFormat.setInputPaths(job, inputDir);

job.setMapperClass(SimpleReduceTransformingMapper.class);
job.setMapOutputValueClass(LongWritable.class);
job.setMapOutputKeyClass(LongWritable.class);

/** Force the reduce to take text as the output value class,
  * instead of the default. */
job.setOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);
job.setReducerClass(SimpleReduceTransformingReducer.class);

/** Cause the keys to be grouped by 10s. */
job.setOutputValueGroupingComparator(GroupByLongGroupingComparator.class);
job.setNumReduceTasks(1);    /** Ensure that all keys go to 1 reduce so
   * the group by is stable. */
```

The following command will run the `SimpleReduce` job (your output will vary slightly):

```
% HADOOP_CLASSPATH=/misc/HadoopSource/commons-lang-2.4.jar ➥
bin/hadoop jar /misc/HadoopSource/hadoop-0.19.0/hadoopprobook.jar ➥
com.apress.hadoopbook.examples.ch5.SimpleReduce -libjars ➥
/misc/HadoopSource/commons-lang-2.4.jar
```

```
Total input paths to process : 5
Running job: job_200902221346_0079
 map 0% reduce 0%
 map 20% reduce 0%
 map 60% reduce 0%
 map 80% reduce 0%
 map 100% reduce 0%
 map 100% reduce 100%
Job complete: job_200902221346_0079
Counters: 20
  File Systems
    HDFS bytes read=7103
    HDFS bytes written=2135
    Local bytes read=9006
    Local bytes written=18176
  Job Counters
    Launched reduce tasks=1
    Launched map tasks=5
    Data-local map tasks=5
  Map Input Keys
    Total=500
  Reduce Output Keys
    Total=35
  Map Output Keys
    Total=500
  Reduce Input Keys
    Total=35
  Map-Reduce Framework
    Reduce input groups=35
    Combine output records=0
    Map input records=500
    Reduce output records=35
    Map output bytes=8000
    Map input bytes=7103
    Combine input records=0
    Map output records=500
    Reduce input records=500
The Job is  complete  and  successfull
```

Note how the output keys are multiples of tens. This is the result of the output value grouping. The actual output is the key, the average value, the number of values averaged, the minimum value, the maximum value, and the difference between the minimum and the maximum.

Now you can print the job output (key, average, count, min, max, difference), as follows:

```
% hadoop dfs -cat SampleReduce.ouput/part-00000
```

```
   0     1032312560  10    8929475  2037836662  2028907187
  10      909677971  10   40027932  2084424645  2044396713
  20     1264310186  10  109435752  2002508155  1893072403
  30      984307588  10  112010776  1912518297  1800507521
  40      925589754  10   38065333  1782409589  1744344256
  50      923048786  10  374030611  1725384504  1351353893
  60      908071213  10  255471236  2115349080  1859877844
  70     1068729097  10   63376590  1954205116  1890828526
  80     1216389986  10   40846046  2120059182  2079213136
  90     1119730476  10  289044657  2002422718  1713378061
 100      638218214  10   10905001  1679731545  1668826544
 110     1208679389  10  351936606  1701974468  1350037862
 120      958900520  10  116429037  1686303707  1569874670
 130      871313033  10   52844729  2019468622  1966623893
 140     1328295033  10  111275382  2059113431  1947838049
 150     1038185198  20   47621146  1976756537  1929135391
 160      980833493  20   72608499  2029753820  1957145321
 170      912381685  20   54099516  1961970644  1907871128
 180     1247773207  20   30716232  2116148228  2085431996
 190      875941698  20    6692770  1663091528  1656398758
 200     1051606085  20   18948588  2123342351  2104393763
 210     1207066231  20  160161337  1952936377  1792775040
 220     1327655145  20   75910389  2078268756  2002358367
 230     1148152274  20  273711624  2074598677  1800887053
 240      735579301  20   43456136  2094659831  2051203695
 250     1115493614  20  190919486  1988623879  1797704393
 260     1026999134  20   59805730  2072846822  2013041092
 270     1109366173  20  198612696  2077682368  1879069672
 280      954780820  20   44018855  2107358734  2063339879
 290      778472644  20   22502766  2063051919  2040549153
 300     1032042843  10  292411084  2097164456  1804753372
 310      822060835  10   90530214  2135412572  2044882358
 320      857131707  10  138285402  1675393365  1537107963
 330     1153129237  10  231919805  1799184626  1567264821
 340      851254291  10  135630114  1965837214  1830207100
```

A Reducer That Uses Three Partitions

A variant of the `SimpleReduce.java` example, called `TotalOrderSimpleReduce.java` (available with the rest of this chapter's downloadable code), uses three partitions, rather than just one. This example demonstrates how to use the `InputSampler` class and the `TotalOrderPartitioner` class, as well as some of the interesting errors that will occur if the partitioner and the OutputValueGroupingComparator do not coordinate fully.

In this example, the grouping operator groups by multiples of ten in the key space. The TotalOrderParititioner selects a random sample of the keys and creates three groups that are roughly even in size given the sample of keys. There is no guarantee that an entire group of keys will not be split into multiple partitions.

This application also requires a custom InputFormat, LongLongTextInputFormat, as the input key and the reduce key must be of the same type for the InputSampler. In the previous version, the map input keys are Text and the reduce input keys are LongWritable. Listing 5-11 shows the core of the LongLongTextInputFormat, the RecordReader.next method.

Listing 5-11. *The RecordReader.next Method of the LongLongTextInputFormat*

```
/** Delegated next, read the textual values from the the data source
  * and convert them into LongWritables.
 * @param key The key object to fill with the next record's key
 * @param value The value object to fill with the next record's value
 * @return true if a record was read or false if at EOF
 * @throws IOException
 * @see org.apache.hadoop.mapred.RecordReader#next(java.lang.Object, ➥
java.lang.Object)
 */
public boolean next(LongWritable key, LongWritable value) throws IOException {
    /** Perform the real read. */
    final boolean res = realReader.next(this.key, this.value);
    if (!res) { /** If at eof, we are done. */
        return false;
    }
    /** Attempt to convert the two text values read into LongWritables.
     * If there is an error, throw an IOException.
     */
    try {
        key.set(Long.valueOf(this.key.toString()));
        value.set(Long.valueOf(this.value.toString()));
        return true;
    } catch( NumberFormatException e) {
        throw new IOException("Invalid key, value " + key + ", " + value);
    }
}
```

The code in Listing 5-12 sets up the JobConf object for the TotalOrderParitioner. Note that natural ordering is set to true. As the keys are long values, they are binary comparable. The call to runInputSampler computes the partitioning index and stores it in the file TotalOrderSimpleReduce.index.

Listing 5-12. *TotalOrderPartition Setup, from TotalOrderSimpleReduce.java*

```
job.setInputFormat(LongLongTextInputFormat.class);
FileInputFormat.setInputPaths(job, inputDir);

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

/** Setup for a total order partitioning. */
job.setPartitionerClass(TotalOrderPartitioner.class);
job.setBoolean("total.order.partitioner.natural.order", true);

/** Force reduce to take text as the output value class, instead of the default. */
job.setOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);
job.setReducerClass(SimpleReduceTransformingReducer.class);

/** Cause the keys to be grouped by 10s. */
job.setOutputValueGroupingComparator(GroupByLongGroupingComparator.class);
/** Ensure that all keys go to 3 reduce to demonstrate order based partitioning. */
job.setNumReduceTasks(3);
runInputSampler(job, inputDir.suffix(".index"));
```

The code in Listing 5-13 runs the `InputSampler` to compute and store the index in `indexFile`. The assumption here is that the `JobConf` object `conf` is already correctly set up with the `InputPaths` and `InputReader`. The sampling strategy is to randomly sample the records with a 0.1% chance that any record is chosen. No more than 100 samples and a suggested 10 input splits are to be read.

Listing 5-13. *Running the InputSampler*

```
/** Generate the TotalOrderPartitioner index file for our key space
 *
 * This will sample the input paths set in conf, using the input format reader.
 * The index file location is written to conf.
 *
 * @param conf The Configuration object to use
 * @param indexFile The index file to generate
 * @throws IOException
 */
public void runInputSampler(final JobConf conf, Path indexFile) throws IOException {
    TotalOrderPartitioner.setPartitionFile(conf, indexFile);
    RandomSampler<LongWritable, LongWritable> sampler = new
    InputSampler.RandomSampler<LongWritable,LongWritable>(0.1, 100, 10);
    InputSampler.<LongWritable,LongWritable>writePartitionFile(conf, sampler);
}
```

The following results show that the input for group 150 is split between partition 0 and partition 1, and that the group 220 is split between partition 2 and partition 3. Your results will differ, as random data generation and selection are occurring.

```
HADOOP_CLASSPATH=/misc/HadoopSource/commons-lang-2.4.jar hadoop jar /misc ➥
/HadoopSource/hadoop-0.19.0/hadoopprobook.jar com.apress.hadoopbook.examples.ch5 ➥
.TotalOrderSimpleReduce -libjars /misc/HadoopSource/commons-lang-2.4.jar
```

```
The Job is  complete  and  successfull
Counter Group: File Systems
    HDFS bytes read 8060
    HDFS bytes written  2257
    Local bytes read    9018
    Local bytes written 18488
Counter Group: Job Counters
    Launched reduce tasks   3
    Launched map tasks  5
    Data-local map tasks    5
Counter Group: Reduce Output Keys
    Total   37
Counter Group: Reduce Input Keys
    Total   37
Counter Group: Map-Reduce Framework
    Reduce input groups 37
    Combine output records  0
    Map input records   500
    Reduce output records   37
    Map output bytes    8000
    Map input bytes 7135
    Combine input records   0
    Map output records  500
    Reduce input records    500
```

Let's examine the reduce output data:

```
for a in 0 1 2; do echo part-0000$a; hadoop dfs -cat TotalOrderSimpleReduce ➥
.ouput/part-0000$a; done
```

```
part-00000
   0      1120696448  10    114767562   2024812642   1910045080
  10      1262245737  10    147134609   2118565837   1971431228
  20      1355678543  10    221719466   2058534489   1836815023
  30      1011945955  10     32549345   1964050949   1931501604
  40      1141622277  10     14444296   2091872332   2077428036
  50      1033598416  10    128237459   1923443602   1795206143
  60      1110802460  10    259693362   1904661969   1644968607
```

```
 70       1241399906  10     41832977    2059443669    2017610692
 80       1230683390  10    103825808    2063631220    1959805412
 90       1128499980  10    107614131    2028701766    1921087635
100       1088361665  10    376207299    1832969382    1456762083
110       1332495922  10    332169914    2049937661    1717767747
120        991086606  10     18158041    1954291526    1936133485
130       1020804065  10    117011726    2094067623    1977055897
140        967879564  10     78769539    2041673853    1962904314
150       1236638804   8    401939855    2012038507    1610098652
part-00001
154       1139330738  12     51795064    1954863887    1903068823
160        993478558  20     54628468    2078982662    2024354194
170       1036438744  20    156951559    1983508735    1826557176
180       1101282242  20     42570729    2097760736    2055190007
190       1193146388  20    113670430    2111312959    1997642529
200       1015890669  20    130204162    2104346838    1974142676
210       1234536770  20    105147150    2045372284    1940225134
220       1464315969   8    479100103    2046550989    1567450886
part-00002
224        954658466  12     96604844    1853232282    1756627438
230        964917299  20    116190161    2115557112    1999366951
240       1207841113  20    352735303    2136588979    1783853676
250       1047422883  20    158450293    2047289337    1888839044
260        884844748  20     54670426    1920120397    1865449971
270       1143486218  20    240046014    2139315373    1899269359
280       1345299024  20    267642220    2099770746    1832128526
290        997769299  20     53033105    2114447296    2061414191
300        566836001  10      3288468    1688928276    1685639808
310        871057357  10      2573252    2059752419    2057179167
320        827237669  10    120300136    2091904736    1971604600
330       1034732041  10     72330772    2053586973    1981256201
340        938330142  10     49826875    2145892833    2096065958
```

Combiners

A combiner is a mini-reducer. The purpose of a combiner is to reduce the volume of data that must be passed to the reducer from a map task by summarizing output records that share the same key. A combiner must implement the `Reducer` interface, and the `reduce()` method of the combiner will be called with each output key and all of the output values that share that key. The output of the combiner is what will be sent over the network to the actual reduce task for the job or written to the final output directory, if there is no reduce task configured. The combiner class `reduce()` method must have the same input and output key/value types as the reducer class.

For each call to `output.collect` made by the `map()` method, the framework will route the key/value pair to the applicable partition, based on the result of the `Partitioner.getPartition` call. When all of the map task input has been processed, these partitions are sorted, and each one is passed as input to the combiner. The combiner's `reduce()` method will be called once for each unique key in the partition, and the values will be the set of values that share that key. The output of the combiner will replace that set of original map outputs, ideally with fewer records or smaller records. This is suitable for jobs that are producing summary information from a large dataset.

Caution The combiner must not change the key values, as the map outputs are not re-sorted after the combiner runs. The reduce phase requires the map outputs to be sorted by key.

It is common for the same class that is used in the reduce task to be used for the combiner. However, this practice often leads to difficult-to-diagnose problems. The combiner must only aggregate values, in a manner that is suitable for processing by the actual reducer. The actual reducer has the larger job of producing the final job output. Problems occur when the reducer is modified to provide some change in the job output, and the person doing the modification is unaware that the reducer is also used as a combiner. It is very important that the combiner class not have side effects, and that the actual reducer be able to properly process the results of the combiner.

Tip It not always simple to build a correct combiner. If a job output has problems, try running the job without the combiner to see if the problem persists. If your actual `reduce()` method is nontrivial, do not also use it as a combiner; instead, write a separate object to combine the map outputs.

The classic example of using a combiner is the `org.apache.hadoop.examples.WordCount` example. This MapReduce job reads a set of text input files and counts the frequency of occurrence of each word in the input files. The map phase outputs each word in the file as a key, with the count of 1. There will be one output record for each word in the file. The combiner will aggregate these into a set that contains one output record per unique word in the input, and the value is the number of times the word appeared in the input. Unless the writer has such a large vocabulary that no word is used more than once, the combiner will greatly reduce the number of records to be processed by the reduce phase.

Listings 5-14, 5-15, and 5-16 show the `JobConf` setup and the `map()` and `reduce()` methods from the `WordCount.java` example, The default `InputFormat` is `TextInputFormat`, which returns a `LongWritable` key, the input line number, and a `Text` value, which is the full line from the input file. The `map()` method tokenizes the line and emits a record for each word of the input record, a `Text` and the value `1`, an `IntWritable`. The `reduce()` method simply sums the values and outputs the word as `Text` and the sum of values, an `IntWritable`. By using the `reduce()` method as a combiner, there is a large reduction in the size of each map task output.

Listing 5-14. *The JobConf Setup, from WordCount.java's run Method*

```
conf.setJobName("wordcount");

// the keys are words (strings)
conf.setOutputKeyClass(Text.class);
// the values are counts (ints)
conf.setOutputValueClass(IntWritable.class);

conf.setMapperClass(MapClass.class);
conf.setCombinerClass(Reduce.class);
conf.setReducerClass(Reduce.class);
```

Listing 5-15. *The Core of the map Method, from WordCount.java*

```
public void map(LongWritable key, Text value,
              OutputCollector<Text, IntWritable> output,
              Reporter reporter) throws IOException {
String line = value.toString();
StringTokenizer itr = new StringTokenizer(line);
while (itr.hasMoreTokens()) {
  word.set(itr.nextToken());
  output.collect(word, one);
}
```

Listing 5-16. *The Core of the reduce Method, from WordCount.java*

```
public void reduce(Text key, Iterator<IntWritable> values,
                 OutputCollector<Text, IntWritable> output,
                 Reporter reporter) throws IOException {
int sum = 0;
while (values.hasNext()) {
  sum += values.next().get();
}
output.collect(key, new IntWritable(sum));
```

When the map task has completed and the partitions are sorted, the combiner may run over the partitions and aggregate values, reducing the total number of key/value pairs that must go over the network to the reduce task.

For example, suppose the map partition dataset originally contained the following:

Key	Value
A	1
A	1
The	1
The	1
The	1
Xylophone	1

After the combiner has completed, the map partition dataset would contain these keys and values:

Key	Value
A	2
The	3
Xylophone	1

It is fairly simply to shoot yourself in the foot with a combiner. The combiner must not cause the loss of any information that is needed by the actual reducer. The classic example of this is a reducer that computes the average of the values for each key. If that reducer is also used as a combiner, the information on the number of records involved computing the average will be lost, and the reduce tasks will see only the average values for each key; the final result will be the average of the averages, instead of the actual average. Combiners also must be idempotent, as they may be run an arbitrary number of times by the Hadoop framework over a given map task's output.

File Types for MapReduce Jobs

The Hadoop framework supports text files, binary (sequence) files, and map files, which are actually a pair of sequence files. Let's take a closer look at each of these file types.

Text Files

The Hadoop framework supports a number of textual input files and output files. The input formats support transparent decompression of input files if an input file name ends in one of the recognized compression format suffixes (`.gz`, `.deflate`, `.lzo_deflate`, `.lzo`, and `.bz2`).

The following formats are available for text files:

`TextInputFormat`: This class reads each line of the input split and returns a record composed of the line number as a `LongWritable` key, and the line itself as a `Text` value. The workhorse class that actually produces the key/value pairs is `org.apache.hadoop.mapred.LineRecordReader`. There is only one tunable parameter: the configuration key, `mapred.linerecordreader.maxlength`, which sets the maximum number of characters allowed in a line. The default value is `Integer.MAX_VALUE`, essentially unlimited. The parameter may be adjusted using the `conf.setInt()` method. For example, `conf.setInt("mapred.linerecordreader.maxlength", 1024)` limits the line length to 1,204 characters.

`KeyValueTextInputFormat`: This class reads each line and splits the line into a key/value pair on a tab character. The workhorse class is `org.apache.hadoop.mapred.KeyValueLineRecordReader`. The separator may be configured by setting the configuration key `key.value.separator.in.input.line`. The key and value are both `Text`. If there is no separator found, the value will be an empty string.

`NLineInputFormat`: This format is ideal for using the input data as control information. It guarantees that each input split will be *N* lines long, with one split being the remaining lines. The configuration key `mapred.line.input.format.linespermap` controls the number of lines of input per map task. The default value is `1`. This may be changed using the `conf.setInt()` method. For example, `conf.setInt("mapred.line.input.format.linespermap", 10)` sets the value to `10`. Under the covers, this uses `org.apache.hadoop.mapred.LineRecordReader` to read the input data and produce `LongWritable`, `Text` key/value pairs.

`MultiFileInputFormat`: This is an abstract class that provides a way for a single task to receive multiple input files as the task's input split. This is commonly done for performance tuning. There is substantial time involved in setting up and starting a task, as well as collecting the results. If the input split is small, a substantial portion of the job runtime may be in the setup and teardown of tasks. The developer is responsible for implementing the `getRecordReader()` method. The `org.apache.hadoop.examples MultiFileWordCount` provides an example of a `RecordReader` that handles reading from multiple files.

`TextOutputFormat`: This is the standard textual output format. It basically calls the `toString` method on each key and value, producing a single-line key `SEPARATOR` value ASCII newline for each output record. The `SEPARATOR` is specified by the value of the configuration key `apred.textoutputformat.separator`, which defaults to `TAB`. If the value is `null`, no `SEPARATOR` and no value will be emitted. If key is `null`, `SEPARATOR value` is emitted. The end-of-record character is hard-coded as an ASCII newline character. Compression is supported if configured.

`MultipleTextOutputFormat`: This format allows you to write output records to different files based on the key and value. The test case `org.apache.hadoop.mapred.TestMultipleTextOutputFormat` provides a sample implementation. The Java source to this class is located in `src/test/org/apache/hadoop/mapred/TestMultipleTextOutputFormat.java` in your Hadoop distribution. Using `MultipleTextOutputFormat`, the user has the option of interceding in the selection of an output file for each output key/value pair in several different ways by overriding different methods.

- For map-only jobs, a portion of the input file path may be included in the output path, by setting the value of the configuration key `mapred.outputformat.numOfTrailingLegs`, to a positive integer. The default is no components of the input file path are used. The value +1 worth of components from the right side of the input file are inserted in the output file path before the file name. This happens after the call to `generateFileNameForKeyValue()`. The actual key and value parameters may be modified by overriding the `getActualKey()` and `getActualValue()` methods.
- You can change the final file name or leaf name via the `String generateLeafFileName(String name)` method. The parameter `name` is the original leaf name. The leaf name is normally the `part-XXXXX`, where the `XXXXX` corresponds to the reduce ordinal number, or the map ordinal number if this is a map-only job. (Changing the leaf name is not commonly done.)
- You can change the path to the output file via the `String generateFileNameForKeyValue(K key, V value, String name)` method. The `name` parameter is the result of `generateLeafFileName`. You can construct arbitrary paths out of the key, value, and name. This is the method commonly overridden by developers. The example in Listing 5-17 produces an output file name of the first letter of the key, a dash, and the partition number. If the key were `akey`, and the `name` were `part-00000`, this key/value pair would go to the file `a-part-00000`.

Listing 5-17. *Simple MultipleTextOutputFormat Output File Name Generator*

```
static class KeyBasedMultipleTextOutputFormat extends
     MultipleTextOutputFormat<Text, Text> {
  protected String generateFileNameForKeyValue(Text key, Text v, String name) {

    return key.toString().substring(0, 1) + "-" + name;
  }
}
```

Caution It is critically important to minimize the number of HDFS files that are opened. HDFS, through at least Hadoop 0.19.0, is designed for small numbers of very large files. Opening many small files will bring your cluster to its knees, and may result in catastrophic failure of your job, as well as your HDFS. It is very easy to open hundreds of thousands of files with `MultipleOutputFormats`.

Sequence Files

Sequence files are a binary format for storing sets of serialized key/value pairs. Sequence files support compression, encapsulate the key and value types, and provide validity checksums. They are an ideal format to use for data that is expensive or complex to parse.

The following formats are available for sequence files:

`SequenceFileInputFormat`: The basic workhorse, this format supports splitting and provides the key and value types. If the input file is a map file (described in the next section), the data file is read.

`SequenceFileAsBinaryInputFormat`: This format returns the raw key and value bytes. It returns `BytesWritable` keys and values.

`SequenceFileAsTextInputFormat`: This format returns the key and value as text. It calls the `toString` method on the key and value classes and returns the key/value pair as `Text,Text`.

`SequenceFileInputFilter`: This format returns only specific records from the sequence file. It provides the `static void setFilterClass(Configuration conf, Class filterClass)` method, which supplies a class that is used to determine which records are returned by the `next(key,value)` method on the reader. The `FilterClass` must implement the `SequenceFileInputFilter.Filter` interface and provide a method `boolean accept(Object key)`. Three filters are provided:

- `RegexFilter.setPattern(Configuration conf, String regex)` provides the regular expression to filter keys.
- `PercentFilter.setFrequency(Configuration conf, int frequency)` provides the way of accepting one record in `frequency` records.
- `MD5Filter.setFrequency(Configuration conf, int frequency)` provides a way of selecting only those records that have an MD5 hash that is evenly divisible by `frequency`.

`SequenceFileOutputFormat`: This format writes the serialized key/value records as output. This is the standard sequence file output. The key and value types must be specified via the `conf.setOutputKeyClass()` and `conf.setOutputValueClass()` methods.

`SequenceFileAsBinaryOutputFormat`: This format writes the raw bytes. The key and value types must be `BytesWritable`, and these raw bytes are written as the records.

Map Files

Map files are a pair of sorted sequence files. If a map file named `mymap` is created, there will be a directory `mymap` in HDFS, and two files in `mymap`: `index` and `data`. The `data` sequence file contains the key/value pairs as records, where the records are sorted in key order. The `index` sequence file is key/location information, where location is the location in `data` where the first record containing a key is located.

Map files provide a way to find a particular key, or region of a sorted file, without having to read the entire file. The HBase project (`http://hadoop.apache.org/hbase`) provides a persistent distributed hash table stored in HDFS, using map files as the underlying storage.

When a map file is specified as a job input, the `data` file is used as the actual input. There is not a `MapFileInputFormat` class; the `SequenceFileInputFormat` class is used. The path specified is the path to the directory containing the `index` and `data` files. `SequenceFileInputFormat` will use the `data` file as the input source.

Tip For best performance, it is strongly suggested that all key lookups be performed in the sort order of the underlying map file. HDFS is highly optimized for streaming files sequentially, and does a very poor job of providing low-latency access to random locations within a file.

For the `MapFileOutputFormat`, the value of the configuration key `io.map.index.interval` determines how many records are written to the `data` sequence file between writes to the `index` sequence file. The default is one index entry for every 128 records.

Map files provide the following methods for looking up key/value pairs.

- `void reset()`: Resets the read position to the beginning of the file.
- `WritableComparable midKey()`: Returns the key roughly in the middle of the file.
- `void finalKey(WritableComparable key)`: Reads the final key.
- `boolean seek(WritableComparable key)`: Seeks to the key, or to the first key after it, if it does not exist.
- `boolean next(WritableComparable key, Writable val)`: Reads the next key/value pair.
- `Writable get(WritableComparable key, Writable val)`: Gets the value for key.
- `WritableComparable getClosest(WritableComparable key, Writable val)`: Gets the closest match to the key, searching as `seek`.
- `WritableComparable getClosest(WritableComparable key, Writable val, final boolean before)`: Works like the previously described `getClosest()` method, unless `before` is `true`—in which case the key before is returned.

Compression

The Hadoop framework supports several types of compression and several compression formats. The framework supports the gzip, zip, sometimes LZO, and bzip2 compression codecs. Native libraries are supplied for Linux i386 and x86_64 for gzip, zip, and LZO for some releases. The framework will transparently compress and uncompress most input and output files. Input files are uncompressed when the input file name has a suffix that maps to one of the known codecs, as shown in Table 5-3.

■**Note** LZO is licensed under the GPL. It is incompatible with the Apache license and has been removed from some distributions. I sincerely wish that this will be resolved and that native LZO becomes a standard part of the Hadoop distribution.

Table 5-3. *Compression Codecs and Mapped File Name Suffixes*

Codec	Suffix
`GzipCodec`	`.gz`
`DefaultCodec`	`.deflate`
`LzoCodec`	`.lzo_deflate`
`LzopCodec`	`.lzo`
`Bzip2Codec`	`.bz2`

Codec Specification

The Hadoop framework supports a number of codecs, with native implementations for a smaller number. `GzipCodec`, `LzoCodec`, and the `DefaultCodec` (zip) have native implementations. `Bzip2Codec` has a pure Java implementation. `LzoCodec` may not be available in some releases due to licensing issues. `Bzip2Codec` is available as of Hadoop 0.19.0.

The list of codecs is stored in the configuration under the key `io.compression.codecs`. In Hadoop 0.19, it has the following value:

```
org.apache.hadoop.io.compress.DefaultCodec,org.apache.hadoop.io.compress. ➥
GzipCodec,org.apache.hadoop.io.compress.BZip2Codec
```

If your environment requires additional codecs, the glue interface is `org.apache.hadoop.io.compress.CompressionCodec`. You would then add the class name to the list of codecs in the `io.compression.codecs` value. The selection of a compression codec is a choice between speed and compression rate. LZO is the fastest by far, and produces files about double the size of gzip. The bzip2 compression is the slowest—substantially slower than gzip—and produces files about one half the size of gzip.

Sequence File Compression

Sequence files are binary record-oriented files, where each record has a serialized key and serialized value. The Hadoop framework supports compressing and decompressing sequence files transparently.

Sequence files may be, and generally should be, compressed. The framework will transparently compress at the record level or the block level. The key `io.seqfile.compression.type` controls the record- or block-level compression for sequence files. A value of `BLOCK` requests block-level compression. A value of `RECORD`, the default, specifies record-level compression. A value of `NONE` disables compression.

In general, block-level compression is recommended, because it provides greater data reduction (at the expense of individual key access). The compression overhead is less, and the compression ratio is much greater. For sequence files that are being used as input to a map or reduce phase, block-level compression is ideal. Sequence files that were written using transparent compression may be divided into multiple input splits by the framework.

Many sites will set the default to `BLOCK` in their `hadoop-site.xml` file, as follows:

```
<property>
  <name> io.seqfile.compression.type</name>
  <value>BLOCK</value>
  <description>Force the default sequence file compression to
           be block compression for efficiency reasons
</value>
</property>
```

Map Task Output

The intermediate map task outputs are a set of sequence files, one per reduce task. As these files must be transferred across the network, a low-overhead compression type, such as gzip or LZO, can provide a substantial reduction in network traffic for little CPU cost. The blog entry at `http://blog.oskarsson.nu/2009/03/hadoop-feat-lzo-save-disk-space-and.html` has some interesting information about compression CPU and size reductions for different Hadoop codecs. Table 5-4 summarizes the compression speed results. For pretty decent compression `LzoCodec` provides high throughput.

■ **Note** I have spent some time running the same job with different compression codecs and `RECORD` or `BLOCK` set for compression, to determine which combination gave the overall performance for the job. At present, this must be done manually.

Table 5-4. *Compression Timings for Hadoop Compression Codecs*

Compressor	Original Size	Compressed Size	Compression Speed	Decompression Speed
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s
LZO—best	8.3GB	2GB	4MB/s	60.6MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

Map output block-level compression may be specified by the job or in the site configuration. If compressed, map output, destined for a reduce task, is always `BLOCK` compressed. Listing 5-18 provides an XML block suitable for inclusion in the `conf/hadoop-site.xml` file to make LZO compression the default for the map task outputs.

Listing 5-18. *A hadoop-site.xml Specification for Map Output Level Compression with LZO*

```
<property>
  <name>mapred.compress.map.output</name>
  <value>true</value>
  <description>Should the outputs of the maps be compressed before being
               sent across the network. Uses SequenceFile compression.
  </description>
</property>

<property>
  <name>mapred.map.output.compression.codec</name>
  <value>org.apache.hadoop.io.compress.LzoCodec</value>
  <description>If the map outputs are compressed, how should they be
               compressed? Use Lzo fast even though not as good compression.
  </description>
</property>
```

Listing 5-19 demonstrates configuring a cluster to always use compression for final output files, and if the final output file is a sequence file, to use `BLOCK` compression.

Listing 5-19. *A hadoop-site.xml Specification for Final Output Files to be Compressed with LZO, and If Sequence Files, BLOCK-Compressed*

```
<property>
  <name>mapred.output.compress</name>
  <value>true</value>
  <description>Should the job outputs be compressed?
  </description>
</property>
```

```
<property>
  <name>mapred.output.compression.type</name>
  <value>BLOCK</value>
  <description>The type of compression to use for final
   output sequence files. May be BLOCK, RECORD or None.
  </description>
</property>

<property>
  <name>mapred.output.compression.codec</name>
  <value>org.apache.hadoop.io.compress.LzoCodec</value>
  <description>If the job outputs are compressed, how should they be
               compressed? Use Lzo fast even though not as good compression.
  </description>
</property>
```

Listings 5-20 and 5-21 demonstrate specifying the compression codec and type via settings on the `JobConf` object.

Listing 5-20. *Setting Intermediate Map Output Compression via the JobConf*

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

Listing 5-21. *Setting Final Output Compression via the JobConf*

```
FileOutputFormat.setOutputCompress (conf, true);
FileOutputFormat.setOutputCompressorClass(LzoCodec.class);
SequenceFileOutputFormat.setOutputCompressionType(conf,CompressionType.BLOCK);
```

JAR, Zip, and Tar Files

The Hadoop framework knows how to unpack JAR, zip, and tar files, but this is only automatically done for archives passed via the `DistributedCache` object The class `org.apache.hadoop.fs.FileUtil` provides two static methods that may be used to unpack these files: `unTar()` for tar files and `unzip()` for zip files. The archives may be unpacked only onto the native file system, not into HDFS.

Summary

The Hadoop Core framework provides a rich set of tools to support a variety of use cases. As with most powerful tools, using them effectively requires training and experience. This chapter has provided a solid foundation for configuring jobs to run successfully and building classes that will actual perform the work for the job.

The effective use of counters in the map and reduce methods provides both the application writer and the organization with metrics for job performance. The `DistributedCache` object provides a way of distributing required data to all of the tasks, without needing to have the data already available on the TaskTracker nodes. You can choose from a variety of input and output formats. The use of compression can greatly reduce the wall clock runtime of a job, as can the use of a combiner. The `KeyFieldBasedComparator` and `KeyFieldBasedPartitioner` classes allow you to implement a secondary sort via the `OutputValueGroupingComparator`. Partitioning is a simple controllable process. You also know how to use `MultipleTextOutputFormat`, and the potential problems it can bring. It is now time to have fun writing MapReduce jobs!