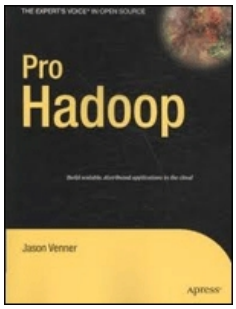

Pro Hadoop

by Jason Venner

Chapter 8: Advanced and Alternate MapReduce Techniques

Overview

This chapter discusses techniques for handling larger jobs with more complex requirements. In particular, the section on map-side joins covers the case in which the input data is already sorted, and the section on chaining discusses ways of adding additional mapper classes to a job without passing all the job data through the network multiple times.

The traditional MapReduce job involves providing a pair of Java classes to handle the map and reduce tasks: reading a set of textual input files using `KeyValueTextInputFormat` or `SequenceFileInputFormat`, and writing the sorted results set out using `TextOutputFormat` or `SequenceFileOutputFormat`. The framework will schedule the map tasks if possible so that each map task's input is local, and provides several ways of reducing the volume of output that must pass over the network to the reduce tasks. This is a good pattern for many (although not all) applications.

There are other options available to Hadoop Core users, either by changing the pattern of the job or by providing the ability to use other languages, such as C++ or Perl and Python, for mapping and reducing.

Streaming: Running Custom MapReduce Jobs from the Command Line

The streaming API allows users to configure and submit complete MapReduce jobs using the command line. As an added bonus, streaming provides the ability to use external programs as any of the job's mapper, combiner, or reducer. The job is a traditional MapReduce job, with the framework handling input splitting, scheduling map tasks, scheduling input split pairs to run, shuffling and sorting map outputs, scheduling reduce tasks to run, and then writing the reduce output to the Hadoop Distributed File System (HDFS).

In the following example, we will demonstrate how to run a simple streaming job to sort all the input records of a dataset using MapReduce. The argument informs the `bin/hadoop` script that the streaming JAR is to be used. Mapper and reduce are defaulted to the identity versions. `-inputformat org.apache.hadoop.mapred.KeyValueTextInputFormat` causes the `KeyValueTextInputFormat` class to be used to provide the key/value pairs from the input. The use of this input format requires the output key format be set to Text via `-D mapred.output.key.class=org.apache.hadoop.io.Text`.

```
bin/hadoop jar ./contrib/streaming/hadoop-0.19.0-streaming.jar -D➥
mapred.output.key.class=org.apache.hadoop.io.Text➥
-inputformat org.apache.hadoop.mapred.KeyValueTextInputFormat➥
-numReduceTasks 2 -input words -output next
```

```
packageJobJar: [/tmp/hadoop-0.19.0-jason/hadoop-unjar11738/]➥
[] /tmp/streamjob11739.jar tmpDir=null
mapred.FileInputFormat: Total input paths to process : 1
streaming.StreamJob: getLocalDirs(): [/tmp/hadoop-0.19.0-jason/mapred/local]
streaming.StreamJob: Running job: job_200902221346_0136
streaming.StreamJob: To kill this job, run:
streaming.StreamJob: /home/jason/src/hadoop-0.19.0/bin/../bin/hadoop job➥
-Dmapred.job.tracker=cloud9:8021 -kill job_200902221346_0136
streaming.StreamJob: Tracking URL: ➥
http://192.168.1.2:50030/jobdetails.jsp?jobid=job_200902221346_0136
streaming.StreamJob:  map 0%  reduce 0%
streaming.StreamJob:  map 50%  reduce 0%
streaming.StreamJob:  map 100%  reduce 0%
streaming.StreamJob:  map 100%  reduce 8%
streaming.StreamJob:  map 100%  reduce 17%
streaming.StreamJob:  map 100%  reduce 58%
streaming.StreamJob:  map 100%  reduce 100%
streaming.StreamJob: Job complete: job_200902221346_0136
streaming.StreamJob: Output: next
```

Hadoop streaming provides the user with the ability to use arbitrary programs for a job's map and reduce methods. The framework handles a streaming job like any other MapReduce job. The job might specify that an executable be used as the map processor and for the reduce processor. Each task will start an instance of the applicable executable and write an applicable representation of the input key/value pairs to the executable. The standard output of the executable is parsed as

textual key/value pairs. The executable being run for the reduce task will given an input line for each value in the reduce value iterator, composed of the key and that value.

The following example uses `/bin/cat` as the mapper and a Perl program to produce line counts of distinct lines from the input set. The argument `-file/tmp/wordCount.pl` causes the file `/tmp/wordCount.pl` to be copied into HDFS and then made available to the map and reduce tasks in their current working directory. The argument `-reducer "/usr/bin/perl -w wordCount.pl"` causes the Perl program `wordCount.pl` to be used to perform the reduce.

```
bin/hadoop jar ./contrib/streaming/hadoop-0.19.0-streaming.jar -input words➥
-output next -file /tmp/wordCount.pl -mapper /bin/cat➥
-reducer "/usr/bin/perl -w wordCount.pl"
```

```
packageJobJar: [/tmp/wordCount.pl, /tmp/hadoop-0.19.0-jason/hadoop-unjar57851/]➥
[] /tmp/streamjob57852.jar tmpDir=null
09/03/15 15:20:40 INFO mapred.FileInputFormat: Total input paths to process : 1
09/03/15 15:20:40 INFO streaming.StreamJob: getLocalDirs():➥
[/tmp/hadoop-0.19.0-jason/mapred/local]
09/03/15 15:20:40 INFO streaming.StreamJob: Running job: job_200902221346_0139
09/03/15 15:20:40 INFO streaming.StreamJob: To kill this job, run:
09/03/15 15:20:40 INFO streaming.StreamJob: ➥
/home/jason/src/hadoop-0.19.0/bin/../bin/hadoop job➥
-Dmapred.job.tracker=cloud9:8021 -kill job_200902221346_0139
09/03/15 15:20:40 INFO streaming.StreamJob: Tracking URL:➥
http://192.168.1.2:50030/jobdetails.jsp?jobid=job_200902221346_0139
09/03/15 15:20:41 INFO streaming.StreamJob:  map 0% reduce 0%
09/03/15 15:20:53 INFO streaming.StreamJob:  map 50% reduce 0%
09/03/15 15:20:55 INFO streaming.StreamJob:  map 100% reduce 0%

09/03/15 15:21:13 INFO streaming.StreamJob:  map 100% reduce 100%
09/03/15 15:21:14 INFO streaming.StreamJob: Job complete: job_200902221346_0139
09/03/15 15:21:14 INFO streaming.StreamJob: Output: next
```

```
cat /tmp/wordCount.pl
```

```
#! /usr/bin/perl -w

use strict;

# The reduce is just passed the same key with
# each value in the value group, so keep track of
# the last key seen to determine when a new value group has started.

my $lastSeenKey;
my $currentCount = 0;

while( <> ) {
    chomp;

    my $currentKey = $_;

    if ($lastSeenKey && $lastSeenKey ne $currentKey) {
        # emit the record for the previous key
        print $currentCount, "\t", $lastSeenKey, "\n";
        $currentCount = 1;
        $lastSeenKey = $currentKey;

    } else {
        # save currentKey away just in case it hasn't been saved
        $lastSeenKey = $currentKey;
        $currentCount++;

    }
}

# make sure that the count for the last key in the input is emitted
```

```
if ($lastSeenKey) {
    print $currentCount, "\t", $lastSeenKey, "\n";
}
```

Hadoop streaming is a wonderful tool. I had a large dataset composed of many input files in one compression format, and the data needed to be compressed in a different format. The author ran a streaming job, with the map executable set to `/bin/cat` and the minimum input split size set to `Long.MAX_VALUE`, and enabled map output compression of the required type. In a few minutes the cluster had uncompressed and recompressed the data files.

The following streaming example takes input from the directory `words`, uses `/bin/cat` as the map executable, has no reduce phase, and compresses the job output using the `GzipCodec`. The `IdentityMapper` could have been used just as easily, but the use of `/bin/cat` is just plain fun.

```
bin/hadoop jar ./contrib/streaming/hadoop-0.19.0-streaming.jar➥
-D mapred.output.compress=true➥
-D mapred.output.compression.codec=org.apache.hadoop.io.compress.GzipCodec➥
-D mapred.min.split.size=111111111111 -input words➥
-mapper /bin/cat -numReduceTasks 0  -output next
```

```
packageJobJar: [/tmp/hadoop-0.19.0-jason/hadoop-unjar13326/]➥
[] /tmp/streamjob13327.jar tmpDir=null
mapred.FileInputFormat: Total input paths to process : 1
streaming.StreamJob: getLocalDirs(): [/tmp/hadoop-0.19.0-jason/mapred/local]
streaming.StreamJob: Running job: job_200902221346_0125
streaming.StreamJob: To kill this job, run:
streaming.StreamJob: /home/jason/src/hadoop-0.19.0/bin/../bin/hadoop job➥
-Dmapred.job.tracker=cloud9:8021 -kill job_200902221346_0125
streaming.StreamJob: Tracking URL: ➥
http://192.168.1.2:50030/jobdetails.jsp?jobid=job_200902221346_0125
streaming.StreamJob:  map 0% reduce 0%
streaming.StreamJob:  map 100% reduce 0%
streaming.StreamJob: Job complete: job_200902221346_0125
streaming.StreamJob: Output: next
```

```
bin/hadoop dfs -ls next
```

```
Found 2 items
drwxr-xr-x   - jason supergroup          0 2009-03-15 01:59 /user/jason/next/_logs
-rw-r--r--   3 jason supergroup    1496397 2009-03-15 01:59➥
/user/jason/next/part-00000.gz
```

Note The streaming API explicitly forces the output key/value classes to be text. If Java classes are used for the mapper, combiner, or reducer, the `InputFormat` used must produce text key/value pairs, or the `mapred.map.output.key.class` and `mapred.map.output.value.class` configuration key/values must be explicitly set to the class names of the key/value classes via `-D mapred.map.output.key.class=java.class.name` or `-D mapred.map.output.value.class=java.class.name`.

Jython: A Way of Interacting with Java Classes in Python

The Jython Project, `http://www.jython.org/`, provides an implementation of Python written in Java. Not all Python features are available. There are additional language constructs that allow the addition of arbitrary Java classes into the namespace of the Jython applications. There are also additional primitive operators for interacting with native Java types, and a transparent translation between the Java String class and the Python string class.

The Hadoop Core distribution provides a Jython example MapReduce application in `src/examples/python/WordCount.py`. People have good results having Python applications used by Hadoop streaming.

Streaming Command-Line Arguments

The streaming command-line interface provides a rich set of command-line arguments for controlling the execution of your streaming job. The standard Hadoop `GenericOptionsParser` arguments are also supported. Table 8-1 describes the streaming-specific command-line arguments, and Figure 8-1 details how the job records are transformed between records and key/value pairs as the framework passes the records through the steps of a streaming job.

Table 8-1: Streaming Specific Command-Line Arguments

Flag	Value	Description
`-input`	Required	The file or directory to use as input. This flag sets the input locat the MapReduce job. It may be given multiple times to provide m input paths.
`-output`	Required	The directory to use for output. This flag sets the directory that o files will be written to. The directory must not exist prior to job st will be created by the framework for the job.
`-mapper`	`org.apache.hadoop.mapred.lib.IdentityMapper`	A Java class name or an executable file. This flag is used as th mapper for the map tasks.
`-combiner`	None	A Java class name or an executable file. This flag is used as th combiner for the map output.
`-reducer`	`org.apache.hadoop.mapred.lib.IdentityReducer`	A Java class name or an executable file. This flag is used as th reducer for the reduce tasks.
`-file`	None	A file to be made available locally to each task. This flag is ofte pass the executable to be used for the mapper, combiner, or re the tasks. The executable will be stored in the current working d of the task.
`-inputformat`	`org.apache.hadoop.mapred.TextInputFormat`	The class name of the handler that will split and read the input fi provide key/value pairs for the mapper. There is special handlir fully qualified class names of `TextInputFormat`, `KeyValueTextInputFormat`, `SequenceFileInputFo` and `SequenceFileAsTextInputFormat`. The default `TextInputFormat` is most efficient because the individual in are not split into key/value pairs.
`-outputformat`	`org.apache.hadoop.mapred.TextOutputFormat`	The class that will be used write the output files for the job.
`-partitioner`	`org.apache.hadoop.mapred.lib.HashPartitioner`	The class that will be used to determine which reduce any given sent to.
`-numReduceTasks`	1	The number of reduce tasks to run.
`-inputreader`	None	Custom class to read records from input files. This is currently u in the framework for the `org.apache.hadoop.streaming.StreamXmlRecord` which lets splits be defined by a beginning and ending XML tag argument `StreamXmlRecord,begin=BEGIN_STRING,end=END_` will result in input splits composed of the text found in files betwe `BEGIN_STRING` and `END_STRING`. (See Listing 8-1 for an ex
`-cmdenv`	None	Key/value pairs to set in the process environment before startin executable mapper, combiner, or reducer.
`-mapdebug`	None	A script to invoke when a map task fails.
`-reducedebug`	None	A script to invoke when a reduce task fails.
`-verbose`	None	Used to turn on verbose output for the streaming framework.

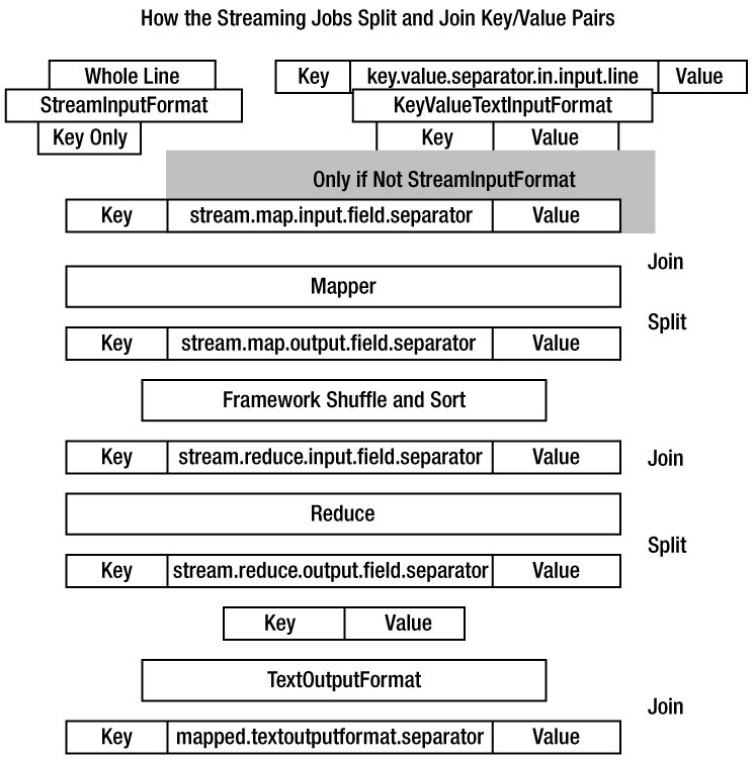

Figure 8-1: How key/value pairs are split and joined in a streaming job

Using -inputreader org.apache.hadoop.streaming.StreamXmlRecordReader

The `-inputreader` command-line flag is an unusual input format handler that provides two core features. The keys that are emitted by the `StreamXMLRecordReader` class contain only the text that is between a beginning and an ending marker, inclusive. Any text in the file that is not between a beginning and an ending marker is ignored.

When run on an Ant build file, the following example produces a sorted list of the target blocks within the XML file. The block selection is based on the `begin=<target name` and `end=</target>` values. The keys produced will contain all the text of the block, starting with `<target name` and ending with `</target>`. This text will include any line separator sequences that are present in the original file in the block. The value is the empty string.

It is best to use `StreamXmlRecordReader` with a Java-based mapper because the key/value pairs may contain line separators. Listing 8-1 demonstrates the use of the `StreamXMLRecordReader`, Listing 8-2 provides the input, and Listing 8-3 provides the output.

Listing 8-1: Using the StreamRecordReader

```
bin/hadoop jar ./contrib/streaming/hadoop-0.19.0-streaming.jar ➥
-Dmapred.mapoutput.key.class=org.apache.hadoop.io.Text -inputreader➥
"org.apache.hadoop.streaming.StreamXmlRecordReader,begin=➥
<target name,end=➥
</target>" -input xml_test.xml -output next
```

```
mapred.FileInputFormat: Total input paths to process : 1
streaming.StreamJob: getLocalDirs(): [/tmp/hadoop-0.19.0-jason/mapred/local]
streaming.StreamJob: Running job: job_200902221346_0144
streaming.StreamJob: To kill this job, run:
streaming.StreamJob: /home/jason/src/hadoop-0.19.0/bin/../bin/hadoop job➥
-Dmapred.job.tracker=cloud9:8021 -kill job_200902221346_0144➥
streaming.StreamJob: Tracking URL: http://192.168.1.2:50030/➥
jobdetails.jsp?jobid=job_200902221346_0144➥
streaming.StreamJob:  map 0%  reduce 0%
streaming.StreamJob:  map 50%  reduce 0%
streaming.StreamJob:  map 100%  reduce 0%
streaming.StreamJob:  map 100%  reduce 100%
```

```
streaming.StreamJob: Job complete: job_200902221346_0144
streaming.StreamJob: Output: next
```

Listing 8-2: StreamXMLRecordReader Sample Input, xml_text.xml

```
<xml>
<target name="part 05">part 05</target>
<target name="part 04"><name>part 04</name></target>
<target name="part 03"><name>part 03</name><value>More things</value></target>
<target name="part 02">
    <name>part 02</name>
    <description>Multi line
    text block>
    </description>
</target>
<target name="part 01"><name>part 01</name>
</target>
</xml>
```

Listing 8-3: StreamRecordReader Output, Next/Part-00000

```
<target name="part 01"><name>part 01</name>
</target>
<target name="part 02">
    <name>part 02</name>
    <description>Multi line
    text block>
    </description>
</target>
<target name="part 03"><name>part 03</name><value>More things</value></target>
<target name="part 04"><name>part 04</name></target>
<target name="part 05">part 05</target>
```

The `StreamXmlRecordReader` parameters are described in Table 8-2. Two parameters, `begin` and `end`, are required. There is some ability to control how much read ahead is done when looking for a match end.

Table 8-2: Control Parameters for StreamXMLRecordReader

Parameter	Default	Description
`begin`	None	Required, String, or Java regular expression used to match the beginning of a block of interest. The value is interpreted as a regular expression if `slowmatch` is `true`.
`end`	None	Required, String, or Java regular expression used to match the end of a block of interest. The value is interpreted as a regular expression if `slowmatch` is `true`.
`slowmatch`	`False`	Attempts to exclude beginning or ending matches that are in CDATA blocks.
`Maxrec`	50000	Used only when `slowmatch` is `true`. The record reader will look forward only the maximum of `maxrec` or `lookahead` bytes for the end of a CDATA block.
`lookahead`	`2*maxrec`	Used only when `slowmatch` is `true`. The record reader will look forward only the maximum of `maxrec` or `lookahead` bytes for the end of a CDATA block.

It is possible to control the maximum size of a key. If the parameter `slowmatch=true` is provided, the framework will attempt to exclude recognizing the beginning and ending text if they are within a CDATA block. The framework will look ahead only in `lookahead` bytes, which by default is equal to twice `maxrec` bytes, or 50,000.

For more information, look at the excellent tutorial on using streaming at the Hadoop Core web site: `http://hadoop.apache.org/core/docs/current/streaming. html`.

Using Pipes

Hadoop Core provides a set of APIs for use by other languages that allow a reasonably rich interaction with the Hadoop framework. There are libraries available for C++. The C++ interface lends itself to usage by Simplified Wrapper and

Interface Generator (SWIG) to generate other language interfaces.

The usage of the pipes APIs are outside of the scope of this book (refer to the wordcountsimple.cc example in your distribution and the tutorial in the Hadoop wiki: `http://wiki.apache.org/hadoop/C++WordCount`).

SWIG

SWIG (`http://www.swig.org/`) is a tool for building language interfaces to C and C++ code.

The C++ APIs for interacting with MapReduce are located in the directory `src/c++/pipes/api/hadoop` of the Hadoop Core distribution.

Using Counters in Streaming and Pipes Jobs

The framework monitors the standard error stream of the mapper and reducer processes. Any line read from the standard error stream that starts with the string `reporter:` is considered by the framework as an interaction command. As of Hadoop 0.19.0, there are two commands honored: `counter:` and `status:`.

The actual command control string is configurable. The default value is `reporter:`, but the value of `stream.stderr.reporter.prefix` will be used if set. The standard error stream is read via a `LineReader`, and a command must be one whole line as returned by the `LineReader.readLine()` method. We strongly recommend that you follow the same practice with counters in your streaming jobs as you do with regular jobs. A counter record should be emitted for each input record, for each output record, for each record that is invalid, and for each crash or exception when possible. The more detail about the job provided by the counters, the more understandable the job behavior is. A line output to the standard error stream, of the form `reporter:counter:UserCounters,InputLines,1`, will increment a counter `InputLines` in the counter group `UserCounters`. If the job was run with the command-line parameter `-D stream.stderr.reporter.prefix=`*`mylog:`*, the line would be *`mylog:`*`counter:UserCounters,InputLines,1`.

> **Note** The value specified for the `stream.stderr.reporter.prefix` configuration key is the entire prefix string, the framework will use that exact string as the prefix, and the text that comes afterward must be `counter:group,counter,increment`. The colon character is not added by the framework as a separator.

Using the reporter:counter:group,counter,increment Command

A line of the form `reporter:counter:group,counter,increment` is converted by the framework into a call on the `Reporter` object of the following:

```
reporter.incrCounter("group", "counter", increment);
```

The parameter `increment` must be a whole number between `LONG.MIN_VALUE` and `Long.MAX_VALUE`. The `group` and `counter` parameters must not have the comma character in them.

Using the reporter:status:message Command

A line of the form `reporter:status:message` is converted by the framework into a call on the `Reporter` object of the following:

```
reporter.setStatus(message);
```

Alternative Methods for Accessing HDFS

Hadoop Core provides two tools: `libhdfs`, a native shared library, and `fuse-dfs`, built upon `libhdfs` to allow non-Hadoop-aware Java programs to access the HDFS file system.

libhdfs

`libhdfs` provides native access to HDFS for applications that can use it. The library provides application writers with a set of methods for interacting with HDFS. The methods in turn use JNI to actually interact with an embedded Java Virtual Machine (JVM) which actually interacts with HDFS. Table 8-3 provides a summary of the methods available.

Table 8-3: Summary of the Methods Provided by libhdfs

Method Name	Description
`hdfsConnect()`	Connects to a NameNode.
`hdfsDisconnect()`	Disconnects from a NameNode.
`hdfsOpenFile()`	Opens an HDFS file.
`hdfsCloseFile()`	Closes an HDFS file.
`hdfsExists()`	Tests for the existence of an HDFS file.
`hdfsSeek()`	Seeks only a read-only HDFS file.
`hdfsTell()`	Tells the current offset of the open HDFS file.
`hdfsRead()`	Reads a block of data from an HDFS file from the current offset point.
`hdfsPread()`	Reads a block of data from at HDFS file starting at a specified offset.
`hdfsWrite()`	Writes a block of data to an HDFS file.
`hdfsFlush()`	Flushes pending data. This method is subject to underlying support for flush in HDFS (not available through Hadoop 0.19.0).
`hdfsAvailable()`	Notes the number of bytes of data available to read without blocking from the open file.
`hdfsCopy()`	Copies a file from one file system to another.
`hdfsMove()`	Moves a file from one file system to another.
`hdfsDelete()`	Deletes a file from HDFS.
`hdfsRename()`	Renames a file in HDFS.
`hdfsGetWorkingDirectory()`	Returns the working directory of the HDFS file system.
`hdfsSetWorkingDirectory()`	Sets the working directory of the HDFS file system.
`hdfsCreateDirectory()`	Creates a directory in HDFS.
`hdfsSetReplication()`	Sets the replication count for an HDFS file.
`hdfsListDirectory()`	Lists the entries in an HDFS directory.
`hdfsGetPathInfo()`	Gets file status information for a path.
`hdfsFreeFileInfo()`	Frees the returned file status information, the pointer returned by `hdfsListDirectory()` and `hdfsGetPathInfo()`.
`hdfsGetHosts()`	Returns the list of DataNode hostnames where the particular block of the specified path are stored.
`hdfsFreeHosts()`	Frees the pointer returned by `hdfsGetHosts()`.
`hdfsGetDefaultBlockSize()`	Returns the basic HDFS block size.
`hdfsGetCapacity()`	Returns the raw storage capacity of the HDFS file system.
`hdfsGetUsed()`	Returns the raw size of all of the files in the HDFS file system.

`libhdfs` is compiled as part of the normal build process. A Linux i386 version is provided in the distribution in the directory `libhdfs`. If you need to build a custom version of the library or want to experiment with it, you need to force the build system to compile it. The command to cause `libhdfs` to be compiled is the following:

```
ant -Dlibhdfs=1 compile-libhdfs
```

If the `libhdfs` property is not set, Ant will not compile `libhdfs`.

Tip Any application using `libhdfs` must have a set `CLASSPATH` environment variable that includes the `hadoop-<rel>-core.jar` file and the JARs in the `lib` directory of the Hadoop distribution, and a shared library loading path that includes the `libjvm.so` shared library from the Java Development Kit (JDK). If they are not preset, the embedded JVM will either fail to launch or the class loader of the embedded JVM will not be able to load the Hadoop classes required to provide HDFS file service.

fuse-dfs

The `fuse-dfs` application uses `libhdfs` and the Filesystem in Userspace (FUSE) APIs to make HDFS file systems appear to be a locally mounted file system on the host machine. It allows arbitrary programs to access data that is stored in HDFS.

Userspace File Systems

The SourceForge project FUSE, `http://fuse.sourceforge.net/`, provides a set of APIs that allow programs written to those APIs to be mounted as host-level file systems.

There is no prebuilt version of `fuse-dfs` bundled into the distribution. In the `src/contrib` subtree is a package called `fuse-dfs`. The README file in `src/contrib/fuse-dfs/README` provides details and requirements. The i386 version may be compiled via the following:

```
ant compile-contrib -Dlibhdfs=1 -Dfusedfs=1
```

The preceding compile command will populate the directory `build/contrib/fuse_dfs`.

Note The `fuse-dfs` compilation environment will compile only for the i386 OS architecture. If X86_64 is required for 64-bit JVMs, the `OS_ARCH` variable must be manually modified in `src/c++/libhdfs/Makefile` and set to `amd64`.

The `fuse-dfs` package requires a modern Linux kernel with the FUSE module, `fuse.ko`, loaded. To actually mount an HDFS file system, the environment variables listed in Table 8-4 must be set correctly.

Table 8-4: fuse_dfs Required Environment Variables

Variable	Required Element	Used By
`LD_LIBRARY_PATH`	This variable must have the paths to the directories containing the following shared libraries: `libjvm.so` from the Java Runtime Environment (JRE), `libhdfs.so` from the Hadoop distribution, and `libfuse.so` from the system FUSE implementation.	The `fuse_dfs` program to load the required shared libraries.
`JAVA_HOME`	The path to the system JRE or JDK installation.	The `fuse_dfs_wrapper.sh` script to set up the runtime environment for the `fuse_dfs` program.
`OS_ARCH`	This variable determines which version of `libjvm.so` is used. The OS compilation architecture of `libjvm.so`, `libhdfs`, and `fuse_dfs` must be identical. The current choices are `i386` and `amd64`.	The `fuse_dfs_wrapper.sh` script to set up the runtime environment for the `fuse_dfs` program.
`CLASSPATH`	This variable must have the JARs from the `lib` directory of the distribution and the core JAR.	The `libjvm.so` shared library will use this for the classpath of the JVM that is embedded the `fuse_dfs` program.

Note The author has used `fuse_dfs` in Hadoop 0.16.0 successfully. In Hadoop 0.19.0, the FUSE mounts produced corrupted directory listings. `fuse_dfs` appears to work correctly in Hadoop 0.19.1.

Mounting an HDFS File System Using fuse_dfs

After successfully compiling the `fuse_dfs` package via the following command, the directory `build/contrib./fuse_dfs` will be populated:

```
ant compile-contrib -Dlibhdfs=1 -Dfusedfs=1
```

The directory should contain at least the files `fuse_dfs` and `fuse_dfs_wrapper.sh`. The `fuse_dfs_wrapper.sh` script makes some assumptions that are not generally applicable and may not work for most installations without modification. The core configuration requires that the `LD_LIBRARY_PATH` environment variable include the directories that `libjvm.so` and `libhdfs.so` are resident in, and that the `CLASSPATH` has the Hadoop Core JAR and the support JARs present.

Listing 8-4, if run from the Hadoop installation root or with the environment variable `HADOOP_HOME` set to the installation

root, will produce the correct settings for `LD_LIBRARY_PATH` and `CLASSPATH`.

Listing 8-4: Script to Compute the Correct LD_LIBRARY_PATH and CLASSPATH Environment Variables for fuse_dfs, setup_fuse_dfs.sh

```
#! /bin/sh

if [ -z "${HADOOP_HOME}" -a -r bin/hadoop-config.sh ]; then
    (echo "n "This script must run from the hadoop installation"
     echo "directory, or have HADOOP_HOME set in the environment") 1>&2
    exit 1
fi

if [ ! -z "${HADOOP_HOME}" -a -d "${HADOOP_HOME}" ]; then
    cd "${HADOOP_HOME}"
    if [ $? -ne 0 ]; then
        echo "Unable to cd to HADOOP_HOME [$HADOOP_HOME]" 1>&2
        exit 1
    fi
fi


if [ ! -r bin/hadoop-config.sh ]; then
    echo "Unable to find the hadoop-config.sh script" 1>&2
    exit 1
fi

export HADOOP_HOME=$PWD
HADOOP_CONF_DIR="${HADOOP_CONF_DIR:-$HADOOP_HOME/conf}"

if [ -z "${JAVA_HOME}" ]; then
    echo "JAVA_HOME is not set" 1>&2
    exit 1
fi

## Cut from bin/hadoop, to ensure classpath is the same as running installation

if [ -f "${HADOOP_CONF_DIR}/hadoop-env.sh" ]; then
  . "${HADOOP_CONF_DIR}/hadoop-env.sh"
fi


# CLASSPATH initially contains $HADOOP_CONF_DIR
CLASSPATH="${HADOOP_CONF_DIR}"
CLASSPATH=${CLASSPATH}:$JAVA_HOME/lib/tools.jar

# for developers, add Hadoop classes to CLASSPATH
if [ -d "$HADOOP_HOME/build/classes" ]; then
  CLASSPATH=${CLASSPATH}:$HADOOP_HOME/build/classes
Fi

for f in $HADOOP_HOME/hadoop-*-core.jar; do
  CLASSPATH=${CLASSPATH}:$f;
done

# add libs to CLASSPATH
for f in $HADOOP_HOME/lib/*.jar; do
  CLASSPATH=${CLASSPATH}:$f;
done

for f in $HADOOP_HOME/lib/jetty-ext/*.jar; do
  CLASSPATH=${CLASSPATH}:$f;
done

LIBJVM=`find -L $JAVA_HOME -wholename '*/server/libjvm.so' -print | tail -1`
if [ -z "${LIBJVM}" ]; then
    echo "Unable to find libjvm.so in JAVA_HOME $JAVA_HOME" 1>&2
    exit 1
```

```
fi

# prefer the libhdfs in build
LIBHDFS=`find $PWD/libhdfs $PWD/build  -iname libhdfs.so -print | tail -1`
if [ -z "${LIBHDFS}" ]; then
    echo "Unable to find libhdfs.so in libhdfs or build" 1>&2
fi

if [ -z "${LD_LIBRARY_PATH}" ]; then
    LD_LIBRARY_PATH="`dirname "${LIBJVM}"`:`dirname "${LIBHDFS}"`"
else
 LD_LIBRARY_PATH="`dirname "${LIBJVM}"`:`dirname "${LIBHDFS}"`":"${LD_LIBRARY_PATH}"
fi
echo "export CLASSPATH='${CLASSPATH}'"
echo "export LD_LIBRARY_PATH='${LD_LIBRARY_PATH}'"
```

After the runtime environment is correctly configured, the `fuse_dfs` program can be run by using the command-line arguments shown in Table 8-5.

Table 8-5: fuse_dfs Command-Line Arguments

Argument	Default	Suggested Value	Description
`server`	None required	NameNode hostname	The server to connect to for HDFS servers.
`port`	None required	NameNode port	The port that the NameNode listens for HDFS requests on.
`entry_timeout`	60	-	The cache timeout for names.
`attribute_timeout`	60	-	The cache timeout for attributes.
`protected`	None	`/user:/tmp`	The list of exact paths that `fuse_dfs` will not delete or move.
`rdbuffer`	10485760	10485760	The size of the buffer used for reading from HDFS.
`private`	None	None	Allows only the user running `fuse_dfs` to access the file system.
`ro`	N/A	N/A	Mounts the file system read-only.
`rw`	N/A	N/A	Mounts the file system read-write.
`debug`	N/A	N/A	Enables debugging messages and runs in the foreground.
`initchecks`	N/A	N/A	Performs environment checks and logs results on startup.
`nopermissions`	`enabled`	`enabled`	Does not do permission checking; permission checking not supported as of Hadoop 0.19.0.
`big_writes`	None	`enabled`	Configures `fuse_dfs` to use large writes.
`usetrash`	`enabled`	`enabled`	Uses the trash directory when deleting files.
`notrash`	`disabled`	`disabled`	Does not use the trash directory when deleting files. Does not work in Hadoop 0.19.0.

The arguments in Table 8-6 are arguments that are passed to the underlying FUSE implementation, not handled directly by `fuse_dfs`.

Table 8-6: Selected FUSE Command-Line Arguments

Argument	Default	Suggested Value	Description
`allow_other`	None	`enabled`	Allows access to other users.
`allow_root`	None	`disabled`	Allows only root access.
`nonempty`	None	`disabled`	Allows mounts over non-empty file or directory.
`fsname`	None	None	Sets the file system name for `/etc/mtab`.
`subtype`	None	None	Sets file system type for `/etc/mtab`.
`direct_io`	None	None	Uses direct I/O instead of buffered I/O.

`kernel_cache`	None	None	Caches files in kernel.
`[no]auto_cache`	None	None	Enables caching based on modification times (off).

The following command will mount a read-only HDFS file system with debugging on. The `fs.default.name` for the file system being mounted is `hdfs://cloud9:8020`. The mount point for the file system is `/mnt/hdfs`, and the arguments after the `/mnt/hdfs` are passed to the FUSE subsystem. These are reasonable arguments for mounting an HDFS file system:

```
./fuse_dfs -oserver=cloud9 -oport=8020 -oro -oinitchecks -oallow_other /mnt/hdfs➥
-o fsname="HDFS" -o debug
```

It is possible to set up a Linux system so that an HDFS is mounted at system start time by updating the system `/etc/fstab` file with a mount request for an HDFS file system. To set up system-managed mounts via `/etc/fstab`, a script `/bin/fuse_dfs` must be created that sets up the environment and then passes the command-line arguments to the actual `fuse_dfs` program. This script just sets up the CLASSPATH environment variable and the `LD_LIBRARY_PATH` variable as the script in Listing 8-4 does.

A candidate line for use in `/etc/fstab` is to mount an HDFS file system at system initialization time. The mount script, for the `/etc/fstab` entry in Listing 8-5, would be passed four arguments. To actually auto mount, the following could be added: `${HADOOP_HOME}/build/contrib/fuse-dfs/fuse_dfs "$3" "$4" "$1" "$2"`. And the script could be placed in `/bin` (see the script `bin_fuse_dfs` in the examples).

Listing 8-5: A Candidate Mount Line for /etc/fstab to Mount an HDFS File System

```
fuse_dfs#dfs://at:9020 /mnt/hdfs fuse rw,usetrash,allow_other,initchecks 0 0
```

Alternate MapReduce Techniques

The traditional MapReduce job reads a set of input data, performs some transformations in the map phase, sorts the results, performs another transformation in the reduce phase, and writes a set of output data. The sorting stage requires data to be transferred across the network and also requires the computational expense of sorting. In addition, the input data is read from and the output data is written to HDFS. The overhead involved in passing data between HDFS and the map phase, and the overhead involved in moving the data during the sort stage, and the writing of data to HDFS at the end of the job result in application design patterns that have large complex map methods and potentially complex reduce methods, to minimize the number of times the data is passed through the cluster.

Many processes require multiple steps, some of which require a reduce phase, leaving at least one input to the next job step already sorted. Having to re-sort this data may use significant cluster resources.

The following section goes into detail about a variety of techniques that are helpful for special situations.

Chaining: Efficiently Connecting Multiple Map and/or Reduce Steps

New in Hadoop 0.19.0 is the ability to connect several map tasks together in a chain. Prior to the chaining feature, the user was forced to either construct large map methods or run multiple jobs as a pipeline, with all the additional I/O overhead. Figure 8-2 provides a graphical depiction of the flow of key/value pairs through a job that uses chaining.

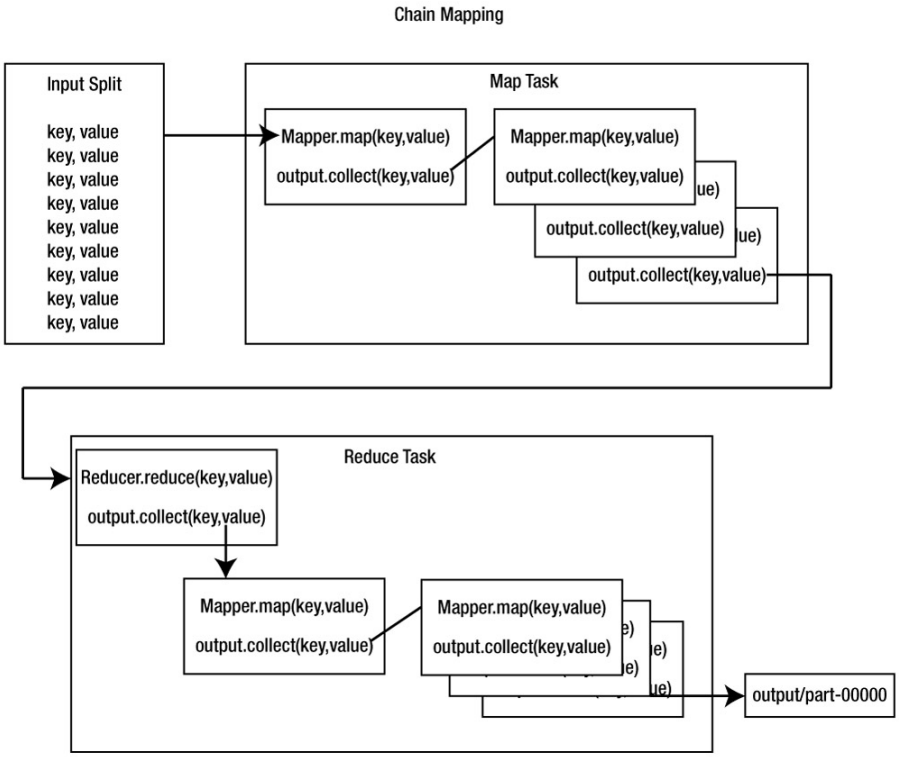

Figure 8-2: Chain mapping

The chaining feature constructs a pipeline, internal to the task, which feeds each key/value pair from each `output.collect` to the map method of the next mapper in the chain. The map task may be a chain, and the reduce task may have a chain as a post processor.

This allows for the construction of simple mapper classes that do one thing well, as well as the ability to rapidly modify a chain to support additional or different features.

Note At least through Hadoop 0.19.0, it is not possible to run the chain mapper through the streaming APIs.

Configuring for Chains

There are two possible chains that can be established for a job: the map task can be a chain or the reduce task can have a chain.

Passing Key/Value Pairs by Value or by Reference

Part of the contract for key/value pair management with a mapper or reducer is that the contents of the key/value are not modified during a call to `output.collect( key, value )`. The framework serializes the key/value into the output format for the particular task, and the `output.collect()` method returns with the contents of the key/value object unchanged.

With chaining, each key/value pair passed to the `output.collect()` method is the input to the next mapper in the chain. During job configuration, when a mapper is being added to a chain, the style of key/value passage is specified, either by value or by reference.

Passing by reference eliminates a serialization and deserialization for the key/value, a potential speed increase. If the `Mapper.map()` method uses the key or value method after the `output.collect()` call, subtle errors may occur if the key or value has been modified by a subsequent mapper.

Note If pass by reference is enabled, some level of verification needs to be in place to ensure that no use of the key/value object is made after a call to `output.collect` or that no mapper in the chain that receives the key or value reference modifies the contents. Any compliance failures in this implicit contract will cause difficulties in diagnosing problems. This is especially difficult because the configuration for pass by reference is remote from the `Mapper.map()` method that has to determine whether the pass by reference is safe and by the fact that the mapper class might be unaware about being part of a chain.

Type Checking for Chained Keys and Values

The standard Hadoop framework verifies that the type of key/value pairs being passed to the map or reduce method are the classes configured for the map or reduce. If they are not, an exception will be thrown.

The `RecordReaders` for the input split will throw an `IOException` for "wrong key class" or "wrong value class", and the `OutputCollector` will throw an `IOException` for "Type mismatch in key from map" or "Type mismatch in value from map".

At least as of Hadoop 0.19.0, the chaining code does not explicitly check the runtime types of the key/value pairs being passed between elements in the chain. The types are checked only during the job configuration phase.

Per Chain Item Job Configuration Objects

The chaining interface provides a way for each item in the chain to receive custom configuration parameters. It is recommended that these custom configurations be light configurations, which have only the special parameters for that item. For a map task, the chain will have only mapper items. For a reduce task, the chain will have a leading reducer item and then some number of mapper items.

At task runtime, a `JobConf` object is made for each item. This `JobConf` object is constructed by making a copy of the localized task `JobConf` object and then copying each key/value pair out of the per map configuration into the copy. This modified copy is then passed to the configure method of the item.

How the close() Method Is Called for Items in a Chain

The mapper `close()` methods are called in order. The reducer close is called after all the mapper `close()` methods have completed. If any `close()` method throws an exception, no further `close()` methods are run.

Caution This is the `close()` behavior, as of Hadoop 0.19.0.

Configuring Mapper Tasks to be a Chain

A mapper task is either a normal map task or a chain. The configuration of one excludes the configuration for the other. A call to the `JobConf.setMapperClass()` method after a chain has been configured will disable the chain.

The framework provides a class, `org.apache.hadoop.mapred.lib.ChainMapper`, which provides the `addMapper()` method. Table 8-7 details the parameters, and Listing 8-6 provides the declaration. The `addMapper()` method configures the mapper tasks to be run as chains and appends the specified mapper class to the end of the current chain mapper task chain.

Table 8-7: The ChainMapper.addMapper Parameters

Type	Parameter	Modified	Description
`JobConf`	`job`	`True`	The per job `JobConf` object.
`Class<? extends Mapper<K1 V1 K2 V2>>`	`klass`	`false`	The mapper class to be run. A `call, job.setMapperClass(ChainMapper.class)`, will be made by this method.
`Class<? extends K1>`	`inputKeyClass`	`false`	The input key class; must be the type of output key of the previous chain item or the type of key for this task if it is the first item in the chain.
`Class<? extends V1>`	`inputValueClass`	`false`	The input value class; must be the type of output value of the previous chain item or the type of value for this task if it is the first item in the chain.
`Class<? extends K2>`	`outputKeyClass`	`false`	The output key class. A call, `job.setMapOutputKeyClass(outputKeyClass)`, will be made in this Method.
`Class<? extends V2>`	`outputValueClass`	`false`	The output value class. A call, `job.setMapOutputValueClass(outputValueClass)`, will be made in this method.
`boolean`	`byValue`	`false`	If `false`, *`klass`*, the mapper class does not use the key or value objects after the call to `output.collect`, or no map later in the chain will modify the values, and the key/value will be passed by reference instead of copied via serialization.
`JobConf`	`mapperConf`	`true`	The configuration object that provides custom configuration data for this

			mapper instance at mapper runtime. The input and output classes will be stored in this object. Any keys present will override the corresponding values in the task's localized `JobConf` object.

Listing 8-6: The ChainMapper.addMapper() method Declaration with JavaDoc

```
/**
   * Adds a mapper class to the chain job's JobConf.
   * <p/>
   * It has to be specified how key and values are passed from one element of
   * the chain to the next, by value or by reference. If a mapper leverages the
   * assumed semantics that the key and values are not modified by the collector
   * 'by value' must be used. If the mapper does not expect this semantics, as
   * an optimization to avoid serialization and deserialization 'by reference'
   * can be used.
   * <p/>
   * For the added mapper the configuration given for it,
   * <code>mapperConf</code>, have precedence over the job's JobConf. This
   * precedence is in effect when the task is running.
   * <p/>
   * IMPORTANT: There is no need to specify the output key/value classes for the
   * ChainMapper, this is done by the addMapper for the last mapper in the chain
   * <p/>
   *
   * @param job              job's JobConf to add the mapper class.
   * @param klass            the mapper class to add.
   * @param inputKeyClass    mapper input key class.
   * @param inputValueClass  mapper input value class.
   * @param outputKeyClass   mapper output key class.
   * @param outputValueClass mapper output value class.
   * @param byValue          indicates if key/values should be passed by value
   * to the next mapper in the chain, if any.
   * @param mapperConf       a JobConf with the configuration for the mapper
   * class. It is recommended to use a JobConf without default values using the
   * <code>JobConf(boolean loadDefaults)</code> constructor with FALSE.
   */
  public static <K1, V1, K2, V2> void addMapper(JobConf job,
                            Class<? extends Mapper<K1, V1, K2, V2>> klass,
                            Class<? extends K1> inputKeyClass,
                            Class<? extends V1> inputValueClass,
                            Class<? extends K2> outputKeyClass,
                            Class<? extends V2> outputValueClass,
                            boolean byValue, JobConf mapperConf)
```

Note For each `addMapper()` call, the `mapperConf` object should be constructed via `JobConf mapperConf = new JobConf(false)`. This will minimize the possibility that a configuration value will step on a task's localized value. Any custom parameters may then be set on the `mapperConf` object before the `addMapper()` call. The `clear()` method may be used to reset a `mapperConf` for use in the next call to `addMapper()`. As of Hadoop 0.19.0, this parameter should not be passed as a `null` because a full `JobConf` object will be initialized.

Configuring the Reducer Tasks to Be Chains

Configuring the reducer phase is very similar to the configuration of the mapper phase with one additional requirement: the job configuration step must make a call to `ChainReducer.setReducer()` before adding any mappers to the reducer chain. Table 8-8 describes the parameters for `ChainReducer.setReducer()`, and Table 8-9 describes the parameters for `ChainReducer.addMapper()`. The other minor difference is that the `ChainReducer.addMapper()` method must be used in place of the `ChainMapper.addmapper()`method.

Table 8-8: ChainReducer.setReducer Parameters

Type	Parameter	Modified	Description
`JobConf`	job	`true`	The per job `JobConf` object.
`Class<? extends Reducer<K1 V1`	*`klass`*	`falsex`	The reducer class to be run. A call, `job.setReducerClass (ChainReducer.class)`, will be made by this method.

K2 V2>>			
Class<? extends K1>	inputKeyClass	false	The input key class; must be the type of output key of the previous chain item or the type of key for this task if it is the first item in the chain.
Class<? extends V1>	inputValueClass	false	The input value class; must be the type of output value of the previous chain item or the type of value for this task if it is the first item in the chain.
Class<? extends K2>	outputKeyClass	false	The output key class. A call, `job.setOutputKeyClass(outputKeyClass)`, will be made in this method.
Class<? extends V2>	outputValueClass	false	The output value class. A call, `job.setOutputValueClass(outputValueClass)`, will be made in this method.
boolean	byValue	false	If `false`, *`klass`*, the reducer class does not use the key or value objects after the call to `output.collect()`, or no map later in the chain will modify the values, and the key/value will be passed by reference instead of copied via serialization.
JobConf	reducerConf	true	The configuration object that provides custom configuration data for this reducer instance at reducer runtime. A `null` may be passed. The input and output classes will be stored in this object. Any keys present will override the corresponding values in the task's localized `JobConf` object.

Table 8-9: ChainReducer.addMapper Parameters

Type	Parameter	Modified	Description
JobConf	Job	true	The per job `JobConf` object.
Class<? extends Mapper<K1 V1 K2 V2>>	*Klass*	false	The mapper class to be run.
Class<? extends K1>	inputKeyClass	false	The input key class; must be the type of output key of the previous chain item or the type of key for this task if it is the first item in the chain.
Class<? extends V1>	inputValueClass	false	The input value class; must be the type of output value of the previous chain item or the type of value for this task if it is the first item in the chain.
Class<? extends K2>	outputKeyClass	false	The output key class. A call, `job.setOutputKeyClass(outputKeyClass)`, will be made in this method.
Class<? extends V2>	outputValueClass	false	The output value class. A call, `job.setOutputValueClass(outputValueClass)`, will be made in this method.
boolean	byValue	false	If `false`, *`klass`*, the mapper class does not use the key or value objects after the call to `output.collect()`, or no map later in the chain will modify the values, and the key/value will be passed by reference instead of copied via serialization.
JobConf	mapperConf	true	The configuration object that provides custom configuration data for this mapper instance at mapper runtime. The input and output classes will be stored in this object. Any keys present will override the corresponding values in the task's localized `JobConf` object.

Note It is important to use `ChainMapper.addMapper()`, `ChainReducer.setReducer()`, and `ChainReducer.addMapper()` instead of the public methods `Chain.addMapper()` and `Chain.setReducer()`. The chain methods do not configure the job level chaining configuration parameters.

The provided code in `ChainMappingExample` and `ChainMappingExampleMapperReducer` provides a simple example of chain mapping that is structured to help you understand the order of events in your chain. The sample code sets a chain mapping job. The maps and the reduce have a particular id to help distinguish them. The actual ordering information has to be extracted from the job log.

Table 8-10 demonstrates running the `ChainMappingExample` and details the exact sequence of the method invocation on the mapper and reducer classes. The assumptions are that the `hadoopprobook` and `commons-lang` JARs are in the current working directory. The construct `2>&1` forces the standard error output to go to the same descriptor as the standard output.

Table 8-10: Event Ordering in the ChainMappingExample

Sequence Number	Task	Method
0	Master map	`constructor()`
1	Master map	`configure()`
2	Map 1	`constructor()`
3	Map 1	`configure()`
4	Map 2	`constructor()`
5	Map 2	`configure()`
6	Master map	`map()` Key0
7	Map 1	`map()` Key0
8	Map 2	`map()` Key0
9	Master map	`map()` Key1
10	Map 1	`map()` Key1
11	Map 2	`map()` Key1
12	Master map	`map()` Key2
13	Map 1	`map()` Key2
14	Map 2	`map()` Key2
15	Master map	`map()` Key3
16	Map 1	`map()` Key3
17	Map 2	`map()` Key3
18	Master map	`map()` Key4
19	Map 1	`map()` Key4
20	Map 2	`map()` Key4
21	Master map	`close()`
22	Map 1	`close()`
23	Map 2	`close()`
24	Reduce 1	`constructor()`
25	Reduce 1	`configure()`
26	Reduce 2	`constructor()`
27	Reduce 2	`configure()`
28	Master Reduce	`constructor()`
29	Master Reduce	`configure()`
30	Master Reduce	`reduce()` Key0
31	Reduce 1	`map()` Key0
32	Reduce 2	`map()` Key0
33	Master Reduce	`reduce()` Key1
34	Reduce 1	`map()` Key1
35	Reduce 2	`map()` Key1
36	Master Reduce	`reduce()` Key2
37	Reduce 1	`map()` Key2
38	Reduce 2	`map()` Key2
39	Master Reduce	`reduce()` Key3

40	Reduce 1	`map()` Key3
41	Reduce 2	`map()` Key3
42	Master Reduce	`reduce()` Key4
43	Reduce 1	`map()` Key4
44	Reduce 2	`map()` Key4
45	Reduce 1	`close()`
46	Reduce 2	`close()`
47	Master Reduce	`close()`

```
HADOOP_CLASSPATH=./commons-lang-2.4.jar hadoop jar hadooppprobook.jar➥
com.apress.hadoopbook.examples.ch8.ChainMappingExample -jt local➥
-verbose -logLevel INFO -tsr ALL --deleteOutput 2>&1 | grep ': Event'
```

Map-Side Join: Sequentially Reading Data from Multiple Sorted Inputs

In a traditional MapReduce job, the framework sorts all data for a reduce task before presenting the keys sequentially to the reduce task. If the input data is already sorted, traditional MapReduce requires that the full map shuffle and sort process take place before the reduce task receives the sorted keys.

Map-side joins provide a way for a map task to receive keys in sequential order and to receive all the values associated with each key (very similar to a reduce task). The map task reads the data directly from HDFS and no reduce is needed, which greatly reduces cluster loading. The author processed a dataset that was reduced from 5 hours to 12 minutes by converting the job to use map-side joins.

The map-side join provides a framework for performing operations on multiple sorted datasets. Although the individual map tasks in a join lose much of the advantage of data locality, the overall job gains due to the potential for the elimination of the reduce phase and/or the great reduction in the amount of data required for the reduce.

Caution There are several constraints on when map-side joins may be used, and the cluster loses capability to manage data locality for the map tasks (see Table 8-11). There are also bugs in the join code that cause unpredictable behavior if there are more than 31 tables in a join; see `https://issues.apache.org/jira/browse/HADOOP-5589` and `https://issues.apache.org/jira/browse/HADOOP-5571`.

Table 8-11: Limitations on Datasets Used in Joins

Limitation	Why
All datasets must be sorted using the same comparator.	The sort ordering of the data in each dataset must be identical for datasets to be joined.
All datasets must be partitioned using the same partitioner.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.
The number of partitions in the datasets must be identical.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.
The `InputFormat` must return the input splits in `Partitioner` order.	The `OutputPartitioner` class returns a partition number for each key, which determines the reduce task each key is assigned to. This partition number is commonly used to construct the file name of the reduce output partition, `part-%05d`. The file name is the string `part-`, followed by a 0 padded five-digit number, which is the reduce output partition. At split time, no information is readily available to determine what partition number the split was originally a part of, so the ordinal number in the `InputSplit` array, returned by the `InputSplit[] InputFormat.getSplits()` method, is used as a surrogate for the partition number. For any given key in the `N`th input split returned by an `InputFormat.getSplits` call, if that key could be present in another dataset, it would be present only in the `N`th split returned by that dataset's `InputFormat.getSplits` call.

The author has used map-side joins extensively in large-scale web crawls to eliminate recently crawled URLs from the set of freshly harvested URLs being prepared for fetching.

As of Hadoop 0.19.0, the join package supports full inner and outer joins. All joins are full table scans at present; one optimization currently missing from the join package is the capability to use the indexes supplied with `org.apache.hadoop.io.MapFile` to skip over unneeded records in datasets.

In the following section, the term *dataset* is used to refer to one join item in the set of elements being joined. The dataset can be an actual dataset or the result of a join.

> **Note** As of at least Hadoop 0.19.0, the joins handle only keys that implement `WritableComparable` and values that implement `Writable`. The join framework has not been updated to handle arbitrary key/value classes.

Examining Join Datasets

A join dataset is specified by providing a dataset name and an `InputFormat` class. A dataset is the set of input splits that an `InputFormat` will produce when given the name as an argument. The `mapred.min.split.size` is set to `Long.MAX_VALUE` before the `InputFormat.getSplits()` method is called. The goal is to force the `InputFormat` not to split individual data files, thereby ensuring that each returned split contains the entirety of a single reduce task output, or partition. The directory and the partition files it contains is a dataset. Using the join package imposes the following limitations on your application.

> **Note** The map-side join has no simple way to discover what reduce partition a split was created as. The `InputFormat`'s split routine is called with the minimum split size set to `Long.MAX_VALUE`, under the assumption that this will cause each split returned to be one complete input partition. The map-side join assumes that the `InputSplit` arrays returned by each dataset's `InputFormat.getSplits()` returns the splits, or partitions in the same partition order (i.e., any given single index slice through arrays of splits will return a set of splits in which all the keys in each set belong to the same partition). If this assumption of equivalent ordering is incorrect, the behavior of the map-side join will be incorrect, and this failure will be detectable only by examining the output data.

Under the Covers: How a Join Works

The customary output of a MapReduce job that has a reduce phase is a single directory with `N` files of the form `part-00000` through `part-0*N-1`. When a `FileInputFormat`-based `InputFormat` is given that output directory as input, and the `mapred.min.split.size` is set to `Long.MAX_VALUE`, `N` input splits will be generated?one for each part file or partition.

For `FileInputFormat`-based datasets, the input splits are returned as an array, in partition file name lexical order (e.g., `part-00000` is first in the array, followed by `part-00001`, and so on).

For each dataset specified in the join, the input splits of the dataset are collected. If the number of input splits returned by each dataset's `InputFormat` is not identical, the framework throws an exception of the form `IOException ("Inconsistent split cardinality from child N, Y/Z")` where `N` is the ordinal number of the dataset, per the input specification; `Y` is the expected number of splits or partition; and `Z` is the number of splits provided by the N^{th} dataset's `InputFormat`.

For each single index slice of the `InputSplit` arrays, a `WrappedRecordReader` is constructed. The `WrappedRecordReaderClass` implements the interface `org.apache.hadoop.mapred.join.ComposableRecordReader` and provides the standard `RecordReader` function of `next(K key,V value)`. The set of `ComposableRecordReaders` that are to be used for a particular join are bundled into a `JoinRecordReader`, which also implements the interface `ComposableRecordReader`. The basic `JoinRecordReader.next(key, value)` method returns the keys of the entire set of keys present in the `WrappedRecordReaders` in `OutputComparator` order. The `value` is a `TupleWritable` object, which contains each value associated with the key across the set of `WrappedRecordReaders`, and information about which `WrappedRecordReader` the value originated in. A `JoinRecordReader` can have any `ComposableRecordReader` implementer as one of its inputs; by default, they are `WrappedRecordReaders` and `JoinRecordReaders`.

Each map task is given a `JoinRecordReader` from the outermost join as the task input record reader and receives the key/value sets of the join one by one in the map method. In a simple case, this `JoinRecordReader` will have `N` `WrappedRecordReaders` from slice `N` of the original `InputSplit` arrays. The default outer join behavior will receive each key in the input split set, in the sort order with all the values for that key. The map method behaves very much like a traditional reduce.

Types of Joins Supported

The join framework comes with support for three types of joins: `outer`, `inner`, and `override`. Joins can be made on direct input datasets or on the results of joining input datasets; arbitrary deep nesting of this joining structure is supported.

Inner Join

The *inner join* is a traditional database-style inner join. The map method will be called with a key/value set only if every dataset in the join contains the key. The `TupleWritable` value will contain a value for every dataset in the join.

Outer Join

The *outer join* is a traditional database-style outer join. The map method will be called for every key in the set of datasets being joined. The `TupleWritable` value will contain values for only those datasets that have a value for this key.

Override Join

The *override join* is unusual in that the there will only ever be one value passed to the map method. In the inner and other joins there will be a set of values passed to the map method. The override join maps a call to the map method with each key in the input split set and with that single value from the rightmost input split or join that has a value for the key.

The use of this join style requires that you order your input datasets (from least to most important). For any given key, your map method will be given the value from the most important dataset that contains the key.

Composing Your Own Join Operators

The join framework provides a mechanism for defining additional operators. The configuration key `mapred.join.define.`*`YOUR_OPERATOR`* must be set to the class name of a class that implements the `ComposableRecordReader` interface. The string *`YOUR_OPERATOR`* in the key definition must be replaced with the name of the custom join operation. *`YOUR_OPERATOR`* can then be passed as the `op` parameter to the compose methods that accept `op`, and used anywhere that the predefined operators, `inner`, `outer`, and `override`, are used.

Details of a Join Specification

A join specification is an operator and a set of data sources. The predefined operators are `inner`, `outer`, and `override` to correspond with the join types. A data source is either a table statement or a join specification. A table statement is a string `tbl(input.format.class.name,"path")`. The comma character is used to separate data sources; parentheses, `()`, are used to group the data sources for an operator. Table 8-12 provides examples of several join data source specifications.

Table 8-12: Data Source Examples

Data Source	Description
`tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat,"textSource")`	A data source located in or at the path `textSource` that contain records that are to be parsed by the `KeyValueTextInputFormat` class.
`inner(tbl (org.apache.hadoop.mapred.SequenceFileAsTextInputFormat,"sequence"),tbl (org.apache.hadoop.mapred.KeyValueTextInputFormat,"textSource"))`	A data source composed of the inner join of the data in sequence file format at or in `sequence`, and the textual data at or in `textSource`. The key/value classes read from `sequence` are converted into `Text`.
`override(inner(tbl (org.apache.hadoop.mapred.SequenceFileAsTextInputFormat,"sequence"),tbl (org.apache.hadoop.mapred.KeyValueTextInputFormat,"textSource")), tbl (org.apache.hadoop.mapred.KeyValueTextInputFormat,"priority")`	A composite data source composed of a nested inner join of `sequence` and `textSource`, joined with `priority`, and with a preference for values from `priority` if multiple sources in the join have values for a given key.

Handling Duplicate Keys in a Dataset

For a join in which a table in the join has duplicate key/value pairs, the map method will be called one time for each possible permutation of the key/value pairs. For example, suppose that a join of two tables is made. Table 1 has two records (`1,a` and `1,b`), and Table 2 has one record (`1, c`). The map method will be called twice with key `1`; once with a tuple `a,c`; and once with a tuple `b,c`.

Composing a Join Specification

The framework provides three helper methods, all named `compose()`, which build either a full join specification or build the input specification for a particular dataset in the join. Two of the methods construct a full join specification and are used when all the datasets within the join have the same `InputFormat`. These two methods differ only in accepting `String` or `Path` objects for the dataset locations. The third is used to construct a table statement for a dataset that includes a specified `InputFormat` and requires the application developer to aggregate the results into a full join specification. The methods are provided via the `CompositeInputFormat` class.

String CompositeInputFormat.compose(Class<? extends InputFormat> inf, String path)

This method produces a table statement from an input format class object and a path to a dataset. The fully qualified class name of `inf` will be used in the returned table statement. This method does not produce a full join statement. It is commonly used when building a join statement from input datasets that have different input formats. (Refer to Table 8-12, items 2 and 3, for examples of complete join statements.)

Here's a sample use of this method:

```
CompositeInputFormat.compose(KeyValueTextInputFormat.class,"mydata");
```

```
tbl(org.apache.hadoop.KeyValueTextInputFormat,"mydata")
```

String CompositeInputFormat.compose(String op, Class<? extends InputFormat> inf, String… path)

This method produces a full join statement. The resulting string can be stored in the configuration under the key `mapred.join.expr` or used as a nested join within another join statement.

Here's a sample use of this method:

```
CompositeInputFormat.compose( "inner", KeyValueTextInputFormat, ➥
"maptest_a.txt","maptest_b.txt","maptest_c.txt" );
```

```
inner(tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat,"maptest_a.txt"),➥
tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat,"maptest_b.txt"),➥
tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat,"maptest_c.txt"))
```

String CompositeInputFormat.compose(String op, Class<? extends InputFormat> inf, Path… path)

This method is identical to the `String` variant except that `Path` objects instead of `String` objects provide the table paths.

Building and Running a Join

There are two critical pieces of engaging the join behavior: the input format must be set to `CompositeInputFormat.class`, and the key `mapred.join.expr` must have a value that is a valid join specification. Optionally, the mapper, reducer, reduce count, and output key/value classes may be set.

The mapper key class will be the key class of the leftmost data source, and the key classes of all data sources should be identical. The mapper value class will be `TupleWritable` for inner, outer, and user-defined join operators. For the `override` join operator, the mapper value class will be the value class of the data sources.

In Listing 8-7, note that the quote characters surrounding the path names are escaped.

Listing 8-7: Synthetic Example of Configuring a Join Map Job

```
/** All of the outputs are Text. */
conf.setOutputFormat(TextOutputFormat.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setMapperClass(MyMap.class);
/** setting the input format to {@link CompositeInputFormat}
  * is the trigger for the map-side join behavior. */
conf.setInputFormat(CompositeInputFormat.class);
conf.set("mapred.join.expr",
"override(tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat, ➥
\"maptest_a.txt\"),tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat, ➥
\"maptest_b.txt\"),tbl(org.apache.hadoop.mapred.KeyValueTextInputFormat, ➥
\"maptest_c.txt\"))");
```

Synthetic Example of Configuring a Join Map Job Using the Compose Helper

```
/** All of the outputs are Text. */
conf.setOutputFormat(TextOutputFormat.class);
conf.setOutputKeyClass(Text.class);
conf.setOutputValueClass(Text.class);
conf.setMapperClass(MyMap.class);
/** setting the input format to {@link CompositeInputFormat}
  * is the trigger for the map-side join behavior. */
conf.setInputFormat(CompositeInputFormat.class);
conf.set("mapred.join.expr",CompositeInputFormat.compose("override",➥
KeyValueTextInputFormat.class, "maptest_a.txt",➥
"maptest_b.txt", "maptest_c.txt"));
```

The Magic of the TupleWritable in the Mapper.map() Method

The map method for the inner and outer join has a value class of `TupleWritable`, and each call to the map method presents one join result row. The `TupleWritable` class provides a number of ways to understand the shape of the join result row. Listing 8-8 provides a sample mapper that demonstrates the use of `TupleWritable.size()`, `TupleWriter.iterator()`, `TupleWritable.has()`, and `TupleWritable.get()` methods. Table 8-13 provides a description of these methods.

Table 8-13: TupleWritable Methods for Interacting with the Join Result Row

Method	Argument	Description
`boolean has(int i)`	The ordinal number of a dataset.	Returns `true` if that dataset provides a value to this result row.
`Writable get(int i)`	The ordinal number of a dataset.	Returns the value object that the dataset has provided to this result row. The object returned by `get` will be reinitialized on the next call to get. The application will need to make a copy of the contents before calling `get()` again if the contents need to exist past the next call to `get()`.
`int size()`		Returns the number of datasets in the join. Only the top-level datasets are counted, even if the dataset is the result of many nested joins. This method is used to provide an index limit for loops through the values using `has` and `get. for( int i = 0; i < tuple.size(); i++ ) if ( tuple.has(i) )`…
`Iterator<Writable>iterator ()`		Returns an `iterator` through the values that are present. For any dataset that did not contribute a value to this result record, the `iterator` will skip over that dataset.

Listing 8-8: A Sample Mapper

```
package com.apress.hadoopbook.examples.ch8;

import java.io.IOException;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
```

```
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.join.TupleWritable;

/** A cut down join mapper that does very little but demonstrates
  * using the TupleWritable
 *
 * @author Jason
 *
 */
class CutDownJoinMapper extends MapReduceBase implements
         Mapper<Text,TupleWritable,Text,Text> {

    Text outputValue = new Text();

    @Override
    public void map(Text key, TupleWritable value,
            OutputCollector<Text, Text> output, Reporter reporter)
            throws IOException {
        try {

            /** The user has two choices here, there is an iterator
             * and a get(i) size option.
             * The down side of the iterator is you don't know what table
             * the value item comes from.
             */

            /** Gratuitous demonstration of using the TupleWritable iterator. */
            int valueCountTotal = 0;
            for( @SuppressWarnings("unused") Writable item : value) {
                valueCountTotal++;
            }
            reporter.incrCounter("Map Value Count Histogram", key.toString() +
                                          " " + valueCountTotal, 1);


            /** Act like the Identity Mapper. */
            final int max = value.size();
            int valuesOutputCount = 0;
            for( int i = 0; i < max; i++) {
                if (value.has(i)) {
                    // Note, get returns the same object initialized
                    // to the data for the current get
                    output.collect( key, new Text(value.get(i).toString() ) );
                    valuesOutputCount++;
                }
            }

            assert valueCountTotal == valuesOutputCount :
                      "The iterator must always return the same number of➥
values as a loop monitoring has(i)";
        } catch (Throwable e) {
            reporter.incrCounter("Exceptions", "MapExceptionsTotal", 1);
            MapSideJoinExample.LOG.error( "Failed to handle record for " + key, e);
        }
    }
}
```

Note A dataset may provide a `null` value to a join result record if the dataset is composed only of keys. Using the construct `get(i)==null` will not correctly indicate that dataset `i` did not have the join result record key present; only the call `having(i)` is sufficient.

Aggregation: A Framework for MapReduce Jobs that Count or Aggregate Data

The Hadoop Core framework provides a package for performing data aggregation jobs. This package may conceptually be thought of as Hadoop streaming for statistics. The analogy is incomplete because some code must be written to use the aggregation services. The aggregation services are provided by classes that implement the interface

`org.apache.hadoop.mapred.lib.aggregate.ValueAggregator`. The framework provides a set of aggregator services (see Table 8-14 for descriptions of the predefined aggregator services). The user can define the custom aggregator (see Listing 8-15). Aggregation can be run via Hadoop streaming. The aggregation framework manages the mapper, combiner, and reducer; and the aggregation service produces the correct key/value pairs to pass forward. The user is responsible for parsing the input record and invoking the aggregate service with the record key and count; the record and count are the traditional map task output key/value pairs. Quite often, the key has no meaning for the job and is simply a label for the end user. The count must the textual representation of an object that the aggregator service expects: a number for `DoubleValueSum`, a whole number for the `LongValue` series, an arbitrary string for the `StringValue` series, and a whole number for `UniqueValueCount` and `ValueHistogram`.

Table 8-14: Predefined Aggregation Services

Class	Description	Id	Key Value	Count Value	Example
`DoubleValueSum`	Computes the sum of input values. The input values are expected to be doubles and are summed. A single output record per reduce.	`DoubleValueSum`	Label	The number to accumulate in the sum. The behavior is identical to `LongValueSum.pl`, so the `LongValueSum.pl` example is used	-
`LongValueMax`	Computes the maximum input value. The input values are expected to be longs, and the max value is output. A single output record per reduce.	`LongValueMax`	Label	The number to challenge the current max value with.	`LongMax`
`LongValueMin`	Computes the minimum input value. The input values are expected to be longs, and the min value is output. A single output record per reduce.	`LongValueMin`	Label	The number to challenge the current min value with. The behavior is essentially identical to `LongValueMax`, so the `LongMax.pl` example is used.	-
`LongValueSum`	Computes the long sum of input values. Input values are expected to be longs, and the sum is output. A single output record per reduce.	`LongValueSum`	Label	The number to add to the sum.	`LongSum`
`StringValueMax`	Computes the lexically greatest input value. The values object's `toString()` method is invoked, and the resulting String is compared. The lexically largest is output. A single output record per reduce.	`StringValueMax`	Label	String to challenge the current lexically largest string.	`StringMa`
`StringValueMin`	Computes the lexically least input value. The values object's `toString()` method is invoked and the resulting String is compared. The lexically smallest is output. A single output record per reduce.	`StringValueMin`	Label	String to challenge the current lexically smallest string.	-
`UniqValueCount`	Computes the set of unique input values. The value object's `equals()` method is used to determine equality. The set of unique object is output. The configuration parameter `aggregate.max.num.unique.values`, which defaults to `Long.MAX_VALUE`, limits the number of unique items accumulated. Any new objects encountered in a map or reduce task past this value are discarded.	`UniqValueCount`	Object as a string.	Ignored; 1 is acceptable.	`UniqValu`
`ValueHistogram`	Computes a histogram of the occurrence counts of the unique input values. The input value object's `equals()` method is used to determine equality.	`ValueHistogram`	The object as a string.	The count of times the object occurred in this record; 1 is usually correct.	`LongHis`

The code that the user must supply can be supplied as a streaming mapper or via a Java class.

Aggregation Using Streaming

The user-supplied code must take an input record and return an aggregator record. The aggregator record is textually the `id: key\tcount`, where `id` is the aggregator service id, `key` is an applicable key for the job, and `count` is the appropriate value for `key`?commonly `1`. Listing 8-9 provides a sample Perl mapper that computes the sums of input files that are sets of long values.

Listing 8-9: Perl Streaming Mapper for LongValueSum of Input Files Composed of Long Values, LongSum.pl

```
#! /usr/bin/perl -w

use strict;

eval {
    while(<>) {
        print STDERR "reporter:counter:Map,Input Records,1\n";
        chomp;
        my @parts = split(/\s/, $_); # split on white space
        foreach my $part ( @parts ) {
            print STDERR "reporter:counter:Map,Output Records,1\n";
            print "LongValueSum:SUM\t$part\n";
        }
    }
};
if ($@) {
    print STDERR "reporter:counter:Map,Exceptions,1\n";
}
```

Each reduce task will have a single output value, the key will be the string SUM, and the value will be the sum of all of the long values routed to that reduce task. The streaming command that was used is in Listing 8-10. An input file with white space separated whole numbers must be in `/tmp/numbers`, and the sums will be placed in `/tmp/numbers_sum_output`.

Listing 8-10: The Streaming Command to invoke LongSum.pl

```
bin/hadoop jar contrib/streaming/hadoop-0.19.0-streaming.jar -jt local➥
-fs file:/// -input /tmp/numbers  -output /tmp/numbers_sum_output -verbose➥
-reducer aggregate -mapper LongSum.pl -file /tmp/LongSum.pl
```

> **Note** The reducer is defined as aggregate for the streaming job in Listing 8-10.

If an error shown in Listing 8-11 happens, it generally means that an unrecognized aggregator id has been output by the mapper.

Listing 8-11: Exception Resulting from an Unrecognized Aggregator Service Id

```
java.lang.NullPointerException
    at org.apache.hadoop.mapred.lib.aggregate.ValueAggregatorCombiner.➥
reduce(ValueAggregatorCombiner.java:59)
    at org.apache.hadoop.mapred.lib.aggregate.ValueAggregatorCombiner.➥
reduce(ValueAggregatorCombiner.java:34)

    at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.➥
combineAndSpill(MapTask.java:1106)
    at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.sortAndSpill➥
(MapTask.java:979)
    at org.apache.hadoop.mapred.MapTask$MapOutputBuffer.flush(MapTask.java:832)
    at org.apache.hadoop.mapred.MapTask.run(MapTask.java:333)
    at org.apache.hadoop.mapred.LocalJobRunner$Job.run(LocalJobRunner.java:138)
```

Aggregation Using Java Classes

A Java application that wants to use the Aggregation services must provide a class that implements the class `ValueAggregatorDescriptor`. The framework provides a base class `ValueAggregatorBaseDescriptor` that can be extended. The job must provide a specific implementation of the method `ArrayList<Entry<Text, Text>> generateKeyValPairs(Object key, Object val);`.This method must provide the same service that the Perl examples in the streaming section did. Listing 8-12 provides the Hadoop example of the `AggregateWordCount` implementation. The `generateEntry()` method is provided by the `ValueAggregatorBaseDescriptor` and builds a key of the form `ID:KEY`, where *`ID`* is `countType` and *`KEY`* is a word found in the tokenized variable `line`.

Listing 8-12: Hadoop Example AggregateWordCount's generateKeyValuePairs() Method

```
public ArrayList<Entry<Text, Text>>
     generateKeyValPairs(Object key, Object val) {
      String countType = LONG_VALUE_SUM;
      ArrayList<Entry<Text, Text>> retv = new ArrayList<Entry<Text, Text>>();
      String line = val.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        Entry<Text, Text> e = generateEntry(countType, itr.nextToken(), ONE);
        if (e != null) {
          retv.add(e);
        }
      }
      return retv;
    }
```

The job is launched by calling the `ValueAggregatorJob.createValueAggregatorJob()` method, as shown in Listing 8-13. The command-line arguments accepted in `args` are listed in Table 8-15.

Table 8-15: Command-line Options Handled by the ValueAggregatorJob. createValueAggregatorJob() method.

Ordinal Position	Optional	Default value	Description
0	Required	None	The input directory or file to load input records from.
1	Required	None	The output directory to store results in. As with any MapReduce job, this directory must not exist prior to job start and will be created by the framework for the job.
2	Optional	1	The number of reduce tasks.
3	Optional	textinputformat	May be textinputformat or seq, indicating that the records in argument 0, input, are to be handled using `TextInputFormat` or `SequenceFileInputFormat`.
4	Optional	None	An XML file to load as configuration data.
5	Optional	Empty String	The suffix to append to the job name, which is initialized to `ValueAggregatorJob:` .

Listing 8-13: Launching the AggregatorWordCount Example from AggregatorWordCount.java

```
public static void main(String[] args) throws IOException {
  JobConf conf = ValueAggregatorJob.createValueAggregatorJob(args
      , new Class[] {WordCountPlugInClass.class});
  JobClient.runJob(conf);
```

Specifying the ValueAggregatorDescriptor Class via Configuration Parameters

The Hadoop test class `TestAggregates` provides an example of specifying the `ValueAggregatorDescriptor` class via the configuration instead of using `ValueAggregatorJob.createValueAggregatorJob()`. Listing 8-14 covers the special configuration to use a Java class that implements `ValueAggregatorDescriptor`. The configuration data causes the class `AggregatorTests` to be used. The text `UserDefined` tells the framework that this is a user-defined class. The parameter `aggregator.descriptor.num` tells the framework how many definitions there are. For each descriptor class to be used by the job, a configuration parameter key of the form `aggregator.descriptor.#` is defined, where `#` is the ordinal number of the descriptor class, less than the value of `aggregator.descriptor.num`. The value is the two-part text string, `UserDefined`, and the fully qualified class name, with a comma separating the parts.

Input records are passed this order to each of the defined classes. In Listing 8-14, there is one class because `aggregator.descriptor.num` is set to `1`, and the class is `org.apache.hadoop.mapred.lib.aggregate.AggregatorTests`, the value of `aggregator.descriptor.0`.

Listing 8-14: How TestAggregates Defines a Custom Aggregator Service.

```
    job.setInt("aggregator.descriptor.num", 1);
    job.set("aggregator.descriptor.0",➥
"UserDefined,org.apache.hadoop.mapred.lib.aggregate.AggregatorTests");
```

The framework does not have an example for defining a custom value aggregation service. Such a service would need to implement the `ValueAggregator` interface, and jobs using the custom service would have to provide an implementation of `ValueAggregatorDescriptor.generateValueAggregator()` that understands the id of the implemented service type.

Side Effect Files: Map and Reduce Tasks Can Write Additional Output Files

The Hadoop Core framework assumes that individual map and reduce tasks can be killed with impunity, which allows the use of speculative execution and retrying of failed tasks. The framework achieves this by placing the task output in a per-task temporary directory that is deleted if the task fails or is killed, or committed to the job output if the task succeeds. Prior to Hadoop release 0.19.0, this per-task directory was available under the task configuration key `mapred.output.dir`. As of Hadoop 0.19.0, this directory is a function of the `OutputCommitter` the job is using. The default `OutputCommitter` is the `FileOutputCommitter`, which stores the task local output directory in the configuration key `mapred.work.output.dir`, and a getter is defined as `FileOutputFormat.getWorkOutputDir(JobConf conf)`. The `FileOutputCommitter` class will move all files and directories from a successful tasks work output directory to the job output directory.

Tip Side effect files should have job unique names; the method `FileOutputFormat.getUniqueName (conf,name)` produces unique names. If `fs` is a `FileSystem` object for the job output directory, and `conf` is the `JobConf` object for the task, `FSDataOutputStream sideEffect = fs.create( new Path ( FileOutputFormat.getWorkOutputDir(conf), FileOutputFormat.getUniqueName (conf,"side_effect_file")) );`, will create a uniquely named side effect file in the task temporary directory with a base name of `side_effect_file` and return an `FsDataOutputStream` object to the opened file. As of Hadoop 0.19.0, the actual file name is `side_effect_file_{m|r}_partition`, where `{m|r}` stands for a map or reduce task, and `partition` is the ordinal number of the map or reduce task.

Tasks that want to create additional output files directly can create them in the temporary output directory.

Tasks can create files in this directory, and the files will be part of the final job output when the tasks succeed. The `OutputCommitter` actually commits the files to the actual job output directory.

Handling Acceptable Failure Rates

Hadoop jobs typically process large volumes of data that originates from some other source. This data, commonly called *dirty data*, is often not perfectly compliant with the data specification. It might also be the case that some input records, while compliant, are unanticipated. These data records can cause a map or a reduce task to hang, crash, or otherwise complete abnormally. By default, the framework will retry the failed task, and the entire job will be terminated if the task does not complete after a number of attempts.

Operationally it is not desirable to have a long running job terminated if only a small number of records are causing problems. New in Hadoop 0.19.0 is the ability to specify that a job can succeed even if a specified number of records cannot be processed.

The framework also allows the job to specify what percentage of the map tasks and what percentage of the reduce tasks must succeed for the job to be considered a success. The default is 100% of the map tasks and 100% of the reduce tasks.

In some applications, there is a threshold for good enough that is less than 100%. In an application the author worked with, there was a piece of legacy code that would catastrophically crash every few thousand records. Due to a variety of business reasons, it was decided not to attempt to fix the legacy application, but instead to just accept those failures.

In the real world of large-scale data processing, often the individual data records are not valuable, and the time value of the transformation result of the dataset is high. In these situations it is acceptable to accept some failing records and or some failing tasks and then let the job complete.

Dealing with Task Failure

The Hadoop framework provides four different mechanisms for dealing with task failure:

- At the highest level, the JobTracker keeps track of the number of tasks that have failed on a particular TaskTracker node on a per-job basis. If this number crosses a threshold, `mapred.max.tracker.failures`, that TaskTracker is blacklisted from executing further tasks for the job.
- The next level is the standard method that most users are familiar with: to retry a failed tasks a number of times, `mapred.map.max.attempts`, for map tasks and `mapred.reduce.max.attempts` for reduce tasks. If any task fails more than the respective number of times, the job is terminated. A task isn't actually considered failed by the JobTracker until it has used up all of its retry attempts.
- The framework also allows the job to specify what percentage of the tasks can fail before the job is terminated. This is normally 0%, but the parameters `mapred.max.map.failures.percent` and `mapred.max.reduce.failures.percent` control the allowed failure percentage.
- The job may also specify that bad record skipping is enabled, as described in the next section.

Skipping Bad Records

You can enable bad record skipping by setting `mapred.skip.map.max.skip.records` and/or `mapred.skip.reduce.max.skip.groups` to a positive nonzero value. The actual value specified is the size of the record block that is acceptable to lose. The smaller the number, the more work the framework might need to do to minimize the dropped records.

The configuration parameter `mapred.skip.attempts.to.start.skipping` determines how many times a task can fail before skip processing is enabled. Skip processing requires that the framework keep track of what record is being processed by the task. For streaming jobs and for jobs that consume multiple records to work on groups of records, the framework cannot track the records; the application developer has to assist the framework in this tracking. For maps and reduces, respectively, there are two configuration parameters and two counters that the application must manage:

- The parameters are `mapred.skip.map.auto.incr.proc.count` and `mapred.skip.reduce.auto.incr.proc.count`. The respective parameter must be set to `false` in the job configuration.
- The application must then increment the respective counter, `SkipBadRecords.COUNTER_MAP_PROCESSED_RECORDS` or `SkipBadRecords.COUNTER_REDUCE_PROCESSED_GROUPS`, for each record processed.

A binary search is used to locate the failing record group within the task. It appears that this search is exhaustive and will continue until the number of task failures is exceeded. For small values of these configuration parameters, increasing the number of task retries is required.

> **Tip** The number of retries is controlled by the configuration parameters `mapred.map.max.attempts` and `mapred.reduce.max.attempts` with setters `JobConf.setMaxMapAttempts()` and `JobConf.setMaxReduceAttempts()`.

Capacity Scheduler: Execution Queues and Priorities

New in Hadoop 0.19.0 is the Capacity Scheduler. This feature provides somewhat dedicated resource pools, queuing priority, and pool-level access control. In the public documentation, a resource pool is referred to as a *queue*, so that term will be used in this document as well.

A queue has priority access to a specified percentage of the overall cluster task execution slots. When a cluster has unused task execution slots, a job in a queue can use the idle slots, even though these slots are over the queue's priority capacity. If a job with priority access to these resources is started, the over-priority allocation task slots will be reclaimed as

needed within a specified time interval by killing the tasks executing on them.

A queue may have an explicit list of users allowed to submit jobs to it. The Capacity Scheduler may also have a list of users allowed to manage the queues.

Enabling the Capacity Scheduler

To enable the Capacity Scheduler, the following parameter must be placed in the `hadoop-site.xml` file for the cluster. As of Hadoop 0.19.0, the Capacity Scheduler JAR is not part of the default runtime classpath. The JAR file is located in `contrib/capacity-scheduler/hadoop-0.19.0-capacity-scheduler.jar` and must be put on the framework classpath. Adding this JAR to the `HADOOP_CLASSPATH` by amending the `conf/hadoop-env.sh` script is sufficient.

Listing 8-15 defines two queues, `default` and `one-small-queue`.

Listing 8-15: Enabling Capacity Scheduling, XML Block, in hadoop-site.xml

```
<property>
  <name>mapred.jobtracker.taskScheduler</name>
  <value>org.apache.hadoop.mapred.CapacityTaskScheduler</value>
</property>
<property>
    <name>mapred.capacity-scheduler.reclaimCapacity.interval</name>
    <value>5</value>
    <final>true</final>
    <description>The polling interval to find needed task slots
           that have a freeloader task executiong.</description>
</property>
<property>
    <name>mapred.queue.names</name>
    <value>default,one-small-queue</value>
    <description>The comma separated list of queue names.</description>
    <final>true</final>
</property>
<property>
    <name>mapred.acls.enabled</name>
    <value>false</value>
    <final>true</final>
<description>Are the access control lists enabled,
           for job submission and queue management.</description>
</property>
```

Each queue that the cluster administrator defines must have a configuration block in the `hadoop-site.xml` file. Listing 8-16 defines one queue, `one-small-queue`, with user `jason` and group `wheel` given submission and control permissions. Replace `one-small-queue` with the queue name being configured. These values could be in `hadoop-site.xml`, but the suggested location is in `capacity-scheduler.xml`. Figure 8-3 shows the JobTracker web interface for this queue set.

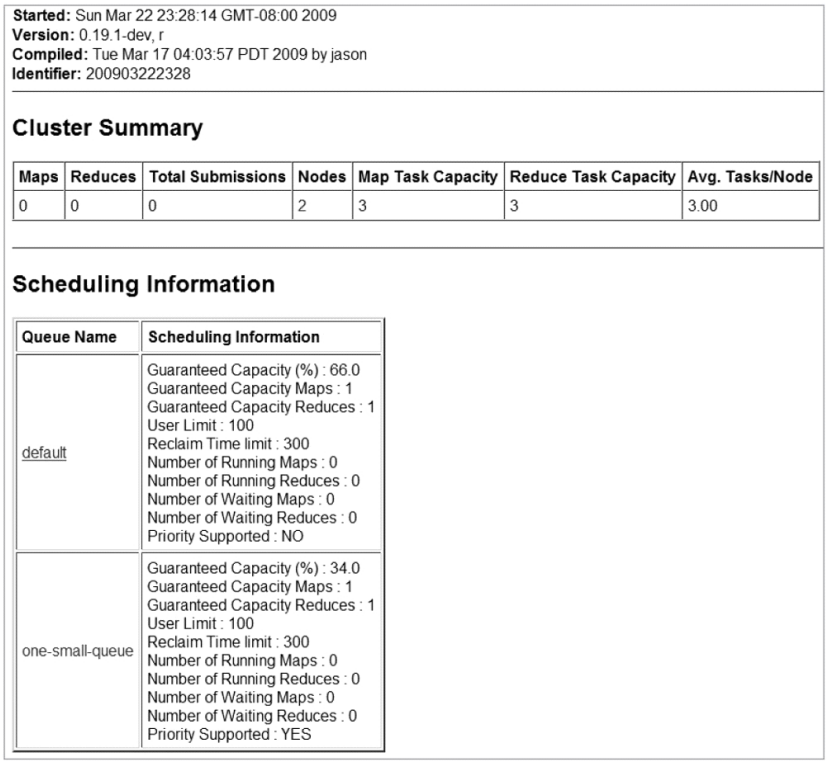

Figure 8-3: Screenshot of a JobTracker screen with two queues enabled

Listing 8-16: For Each Queue to be Defined, XML Block in capacity-scheduler.xml

```
<!-- for each queue, the following set of properties must exist -->
<!--, where one-small-queue is the name of the queue -->
<property>
    <name>mapred.capacity-scheduler.queue.one-small-queue.guaranteed-capacity</name>
    <value>34</value>
    <final>true</final>
    <description>A value between 0 and 100, the percentage
      of the task execution slot that one-small-queue has
            priority for.</description>
</property>

<property>
    <name>mapred.capacity-scheduler.queue.one-small-queue.reclaim-time-limit</name>
    <value>300</value>
    <description>The time in seconds before a task running on a
          loaned out slot is killed when the slot is needed.</description>
    <final>true</final>
</property>
<property>
    <name>mapred.capacity-scheduler.queue.one-small-queue.supports-priority</name>
    <value>true</value>
    <description>If true, the queue supports priorities for queued
                 jobs.</description>
</property>
<property>
    <name>
       mapred.capacity-scheduler.queue.one-small-queue.minimum-user-limit-percent
    </name>
    <value>100</value>
    <description>The percentage of the resources of this queue any user may
                 use at one time.</description>
</property>
<property>
    <name>mapred.queue.one-small-queue.acl-submit-job</name>
    <value>jason wheel</value>
    <final>true</final>
```

```
    <description>Two comma separated lists, separated by a space. The list of
              users and the list of groups. This is the set that may submit
              jobs to one-small-queue</description>
</property>
<property>
    <name>mapred.queue.one-small-queue.acl-administer-job</name>
    <value>jason wheel</value>
    <final>true</final>
    <description>Two comma separated lists, separated by a space. The list of
                 users and the list of groups. This is the set that may kill
                 or change the priority of other users jobs.</description>
</property>
```

The sum of the percentage cluster capacity for all queues must not exceed 100, or the Job-Tracker will not start, and there will be an exception in the log file:

```
org.apache.hadoop.mapred.JobTracker: java.lang.IllegalArgumentException: ➥
Sum of queue capacities over 100% at SOMEVALUE
```

Summary

The Hadoop framework provides a powerful set of tools to enable users to run more than standard MapReduce jobs. This chapter covers a number (but by no means all) of the features. Hadoop is under active development, and new features are being introduced on a regular basis. The Hadoop streaming and aggregator features are powerful and provide the user command-line tools for performing data analysis on large datasets. Chain mapping provides a way to maintain code simplicity and reduce overall data flow through the system by allowing multiple mapper classes to be applied to the data for a job. Map-side joins provide database-style joins that can drastically speed up jobs that process bulk data that is already sorted. There are also a number of features that have become their own Apache projects (see Chapter 10).