

CHAPTER 8

MapReduce Types and Formats

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

MapReduce Types

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

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

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

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

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

```
// ...
}
}
```

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

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

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

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

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

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

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

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

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

```
partition: (K2, V2) → integer
```

Or in Java:

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

MapReduce Signatures in the Old API

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

```
public interface Mapper<K1, V1, K2, V2> extends JobConfigurable, Closeable {  
    void map(K1 key, V1 value,  
             OutputCollector<K2, V2> output, Reporter reporter) throws IOException;  
}  
  
public interface Reducer<K2, V2, K3, V3> extends JobConfigurable, Closeable {  
    void reduce(K2 key, Iterator<V2> values,  
                OutputCollector<K3, V3> output, Reporter reporter) throws IOException;  
}  
  
public interface Partitioner<K2, V2> extends JobConfigurable {  
    int getPartition(K2 key, V2 value, int numPartitions);  
}
```

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

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

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

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

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

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

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

The Default MapReduce Job

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

```
public class MinimalMapReduce extends Configured implements Tool {  
  
    @Override  
    public int run(String[] args) throws Exception {  
        if (args.length != 2) {  
            System.err.printf("Usage: %s [generic options] <input> <output>\n",  
                getClass().getSimpleName());  
            ToolRunner.printGenericCommandUsage(System.err);  
            return -1;  
        }  
  
        Job job = new Job(getConf());  
        job.setJarByClass(getClass());  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
        return job.waitForCompletion(true) ? 0 : 1;  
    }  
  
    public static void main(String[] args) throws Exception {  
        int exitCode = ToolRunner.run(new MinimalMapReduce(), args);  
        System.exit(exitCode);  
    }  
}
```

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

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

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

```
0>0029029070999991901010106004+64333+023450FM-12+000599999V0202701N01591...  
0>0035029070999991902010106004+64333+023450FM-12+000599999V0201401N01181...  
135>0029029070999991901010113004+64333+023450FM-12+000599999V0202901N00821...  
141>0035029070999991902010113004+64333+023450FM-12+000599999V0201401N01181...  
270>0029029070999991901010120004+64333+023450FM-12+000599999V020991C00001...  
282>0035029070999991902010120004+64333+023450FM-12+000599999V0201401N01391...
```

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

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

```
public class MinimalMapReduceWithDefaults extends Configured implements Tool {

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

        job.setInputFormatClass(TextInputFormat.class);

        job.setMapperClass(Mapper.class);

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

        job.setPartitionerClass(HashPartitioner.class);

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

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

        job.setOutputFormatClass(TextOutputFormat.class);

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

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

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

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

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

```

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

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

```

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

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

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

```

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

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

```

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

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

```

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

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

```

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

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

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

Choosing the Number of Reducers

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

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

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

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

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

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

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

The default Streaming job

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

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

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

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

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

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

Keys and values in Streaming

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

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

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

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

Table 8-3. Streaming separator properties

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

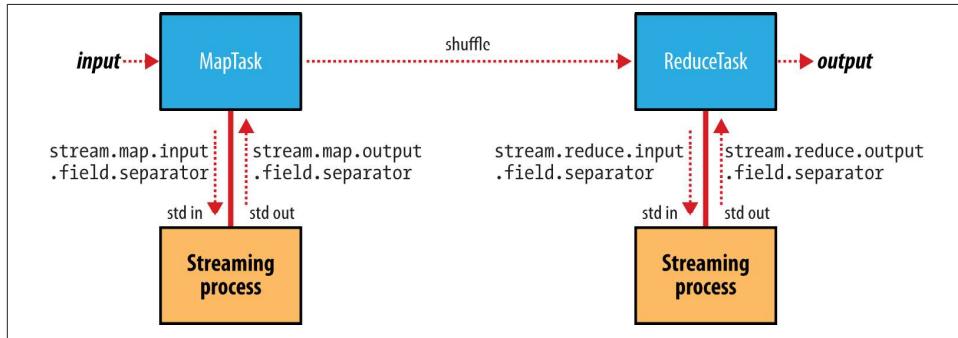


Figure 8-1. Where separators are used in a Streaming MapReduce job

Input Formats

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

Input Splits and Records

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

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

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

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

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

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

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

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

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

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

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

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



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

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

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

FileInputFormat

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

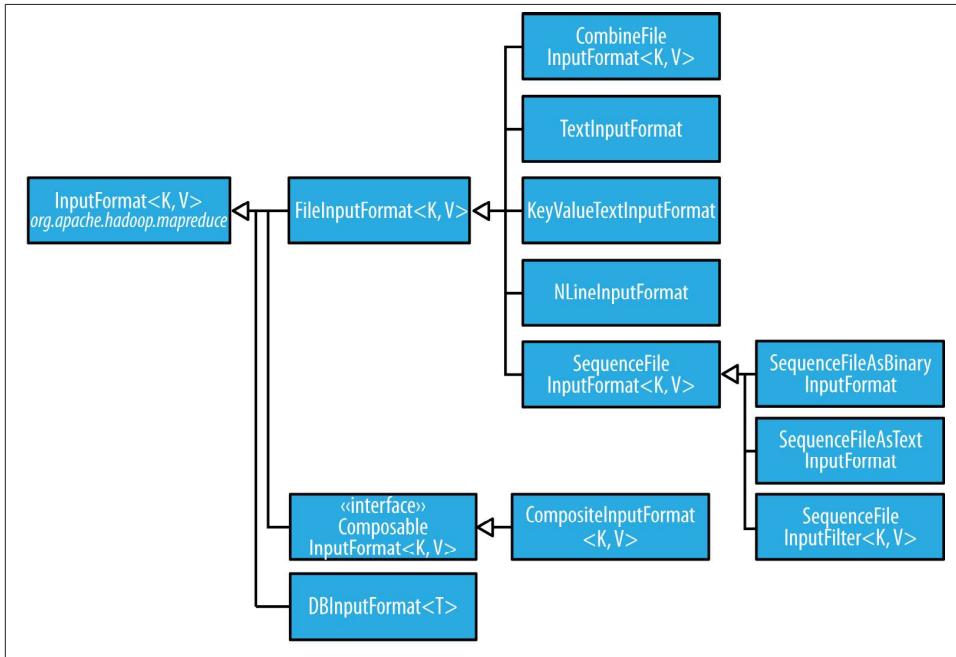


Figure 8-2. *InputFormat* class hierarchy

FileInputFormat input paths

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

```

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

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

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



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

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

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

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

Table 8-4. Input path and filter properties

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

FileInputFormat input splits

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

Table 8-5. Properties for controlling split size

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

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

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

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

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

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

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

and by default:

```
minimumSize < blockSize < maximumSize
```

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

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

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

Small files and `CombineFileInputFormat`

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

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

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



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

Preventing splitting

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

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

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

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

File information in the mapper

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

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

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

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

In addition to the properties in [Table 8-7](#), all mappers and reducers have access to the properties listed in “[The Task Execution Environment](#)” on page 203.

Table 8-7. File split properties

| FileSplit method | Property name | Type | Description |
|--------------------------|---|-----------------|--|
| <code>getPath()</code> | <code>mapreduce.map.input.file</code> | Path/ String | The path of the input file being processed |
| <code>getStart()</code> | <code>mapreduce.map.input.start</code> | long | The byte offset of the start of the split from the beginning of the file |
| <code>getLength()</code> | <code>mapreduce.map.input.length</code> | long | The length of the split in bytes |

In the next section, we’ll see how to use a `FileSplit` when we need to access the split’s filename.

Processing a whole file as a record

A related requirement that sometimes crops up is for mappers to have access to the full contents of a file. Not splitting the file gets you part of the way there, but you also need to have a `RecordReader` that delivers the file contents as the value of the record. The listing for `WholeFileInputFormat` in [Example 8-2](#) shows a way of doing this.

Example 8-2. An InputFormat for reading a whole file as a record

```
public class WholeFileInputFormat
    extends FileInputFormat<NullWritable, BytesWritable> {

    @Override
    protected boolean isSplitable(JobContext context, Path file) {
        return false;
    }

    @Override
    public RecordReader<NullWritable, BytesWritable> createRecordReader(
        InputSplit split, TaskAttemptContext context) throws IOException,
        InterruptedException {
        WholeFileRecordReader reader = new WholeFileRecordReader();
        reader.initialize(split, context);
        return reader;
    }
}
```

`WholeFileInputFormat` defines a format where the keys are not used, represented by `NullWritable`, and the values are the file contents, represented by `BytesWritable` instances. It defines two methods. First, the format is careful to specify that input files should never be split, by overriding `isSplitable()` to return `false`. Second, we

implement `createRecordReader()` to return a custom implementation of `RecordReader`, which appears in [Example 8-3](#).

Example 8-3. The RecordReader used by WholeFileInputFormat for reading a whole file as a record

```
class WholeFileRecordReader extends RecordReader<NullWritable, BytesWritable> {

    private FileSplit fileSplit;
    private Configuration conf;
    private BytesWritable value = new BytesWritable();
    private boolean processed = false;

    @Override
    public void initialize(InputSplit split, TaskAttemptContext context)
        throws IOException, InterruptedException {
        this.fileSplit = (FileSplit) split;
        this.conf = context.getConfiguration();
    }

    @Override
    public boolean nextKeyValue() throws IOException, InterruptedException {
        if (!processed) {
            byte[] contents = new byte[(int) fileSplit.getLength()];
            Path file = fileSplit.getPath();
            FileSystem fs = file.getFileSystem(conf);
            FSDataInputStream in = null;
            try {
                in = fs.open(file);
                IOUtils.readFully(in, contents, 0, contents.length);
                value.set(contents, 0, contents.length);
            } finally {
                IOUtils.closeStream(in);
            }
            processed = true;
            return true;
        }
        return false;
    }

    @Override
    public NullWritable getCurrentKey() throws IOException, InterruptedException {
        return NullWritable.get();
    }

    @Override
    public BytesWritable getCurrentValue() throws IOException,
        InterruptedException {
        return value;
    }

    @Override
```

```

    public float getProgress() throws IOException {
        return processed ? 1.0f : 0.0f;
    }

    @Override
    public void close() throws IOException {
        // do nothing
    }
}

```

`WholeFileRecordReader` is responsible for taking a `FileSplit` and converting it into a single record, with a `null` key and a value containing the bytes of the file. Because there is only a single record, `WholeFileRecordReader` has either processed it or not, so it maintains a Boolean called `processed`. If the file has not been processed when the `nextKeyValue()` method is called, then we open the file, create a byte array whose length is the length of the file, and use the Hadoop `IOUtils` class to slurp the file into the byte array. Then we set the array on the `BytesWritable` instance that was passed into the `next()` method, and return `true` to signal that a record has been read.

The other methods are straightforward bookkeeping methods for accessing the current key and value types and getting the progress of the reader, and a `close()` method, which is invoked by the MapReduce framework when the reader is done.

To demonstrate how `WholeFileInputFormat` can be used, consider a MapReduce job for packaging small files into sequence files, where the key is the original filename and the value is the content of the file. The listing is in [Example 8-4](#).

Example 8-4. A MapReduce program for packaging a collection of small files as a single SequenceFile

```

public class SmallFilesToSequenceFileConverter extends Configured
    implements Tool {

    static class SequenceFileMapper
        extends Mapper<NullWritable, BytesWritable, Text, BytesWritable> {

        private Text filenameKey;

        @Override
        protected void setup(Context context) throws IOException,
            InterruptedException {
            InputSplit split = context.getInputSplit();
            Path path = ((FileSplit) split).getPath();
            filenameKey = new Text(path.toString());
        }

        @Override
        protected void map(NullWritable key, BytesWritable value, Context context)
            throws IOException, InterruptedException {
            context.write(filenameKey, value);
        }
    }
}

```

```

        }

    }

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

        job.setInputFormatClass(WholeFileInputFormat.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);

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

        job.setMapperClass(SequenceFileMapper.class);

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

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

```

Because the input format is a WholeFileInputFormat, the mapper only has to find the filename for the input file split. It does this by casting the InputSplit from the context to a FileSplit, which has a method to retrieve the file path. The path is stored in a Text object for the key. The reducer is the identity (not explicitly set), and the output format is a SequenceFileOutputFormat.

Here's a run on a few small files. We've chosen to use two reducers, so we get two output sequence files:

```
% hadoop jar hadoop-examples.jar SmallFilesToSequenceFileConverter \
-conf conf/hadoop-localhost.xml -D mapreduce.job.reduces=2 \
input/smallfiles output
```

Two part files are created, each of which is a sequence file. We can inspect these with the `-text` option to the filesystem shell:

```
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00000
hdfs://localhost/user/tom/input/smallfiles/a 61 61 61 61 61 61 61 61 61
hdfs://localhost/user/tom/input/smallfiles/c 63 63 63 63 63 63 63 63 63
hdfs://localhost/user/tom/input/smallfiles/e
% hadoop fs -conf conf/hadoop-localhost.xml -text output/part-r-00001
hdfs://localhost/user/tom/input/smallfiles/b 62 62 62 62 62 62 62 62 62
hdfs://localhost/user/tom/input/smallfiles/d 64 64 64 64 64 64 64 64 64
hdfs://localhost/user/tom/input/smallfiles/f 66 66 66 66 66 66 66 66 66
```

The input files were named *a*, *b*, *c*, *d*, *e*, and *f*, and each contained 10 characters of the corresponding letter (so, for example, *a* contained 10 “a” characters), except *e*, which was empty. We can see this in the textual rendering of the sequence files, which prints the filename followed by the hex representation of the file.



There's at least one way we could improve this program. As mentioned earlier, having one mapper per file is inefficient, so subclassing `CombineFileInputFormat` instead of `FileInputFormat` would be a better approach.

Text Input

Hadoop excels at processing unstructured text. In this section, we discuss the different `InputFormats` that Hadoop provides to process text.

`TextInputFormat`

`TextInputFormat` is the default `InputFormat`. Each record is a line of input. The key, a `LongWritable`, is the byte offset within the file of the beginning of the line. The value is the contents of the line, excluding any line terminators (e.g., newline or carriage return), and is packaged as a `Text` object. So, a file containing the following text:

```
On the top of the Crumpetty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.
```

is divided into one split of four records. The records are interpreted as the following key-value pairs:

```
(0, On the top of the Crumpetty Tree)  
(33, The Quangle Wangle sat,)  
(57, But his face you could not see,)  
(89, On account of his Beaver Hat.)
```

Clearly, the keys are *not* line numbers. This would be impossible to implement in general, in that a file is broken into splits at byte, not line, boundaries. Splits are processed independently. Line numbers are really a sequential notion. You have to keep a count of lines as you consume them, so knowing the line number within a split would be possible, but not within the file.

However, the offset within the file of each line is known by each split independently of the other splits, since each split knows the size of the preceding splits and just adds this onto the offsets within the split to produce a global file offset. The offset is usually sufficient for applications that need a unique identifier for each line. Combined with the file's name, it is unique within the filesystem. Of course, if all the lines are a fixed width, calculating the line number is simply a matter of dividing the offset by the width.

The Relationship Between Input Splits and HDFS Blocks

The logical records that `FileInputFormats` define usually do not fit neatly into HDFS blocks. For example, a `TextInputFormat`'s logical records are lines, which will cross HDFS boundaries more often than not. This has no bearing on the functioning of your program—lines are not missed or broken, for example—but it's worth knowing about because it does mean that data-local maps (that is, maps that are running on the same host as their input data) will perform some remote reads. The slight overhead this causes is not normally significant.

Figure 8-3 shows an example. A single file is broken into lines, and the line boundaries do not correspond with the HDFS block boundaries. Splits honor logical record boundaries (in this case, lines), so we see that the first split contains line 5, even though it spans the first and second block. The second split starts at line 6.

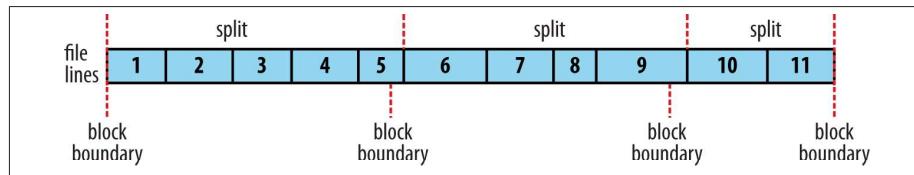


Figure 8-3. Logical records and HDFS blocks for `TextInputFormat`

Controlling the maximum line length. If you are using one of the text input formats discussed here, you can set a maximum expected line length to safeguard against corrupted files. Corruption in a file can manifest itself as a very long line, which can cause out-of-memory errors and then task failure. By setting `mapreduce.input.linerecordreader.line maxlen` to a value in bytes that fits in memory (and is comfortably greater than the length of lines in your input data), you ensure that the record reader will skip the (long) corrupt lines without the task failing.

KeyValueTextInputFormat

`TextInputFormat`'s keys, being simply the offsets within the file, are not normally very useful. It is common for each line in a file to be a key-value pair, separated by a delimiter such as a tab character. For example, this is the kind of output produced by `TextOutputFormat`, Hadoop's default `OutputFormat`. To interpret such files correctly, `KeyValueTextInputFormat` is appropriate.

You can specify the separator via the `mapreduce.input.keyvaluelinerecordreader.key.value.separator` property. It is a tab character by default. Consider the following input file, where → represents a (horizontal) tab character:

```
line1→On the top of the Crumpty Tree  
line2→The Quangle Wangle sat,  
line3→But his face you could not see,  
line4→On account of his Beaver Hat.
```

Like in the `TextInputFormat` case, the input is in a single split comprising four records, although this time the keys are the `Text` sequences before the tab in each line:

```
(line1, On the top of the Crumpty Tree)  
(line2, The Quangle Wangle sat,)  
(line3, But his face you could not see,)  
(line4, On account of his Beaver Hat.)
```

NLineInputFormat

With `TextInputFormat` and `KeyValueTextInputFormat`, each mapper receives a variable number of lines of input. The number depends on the size of the split and the length of the lines. If you want your mappers to receive a fixed number of lines of input, then `NLineInputFormat` is the `InputFormat` to use. Like with `TextInputFormat`, the keys are the byte offsets within the file and the values are the lines themselves.

N refers to the number of lines of input that each mapper receives. With N set to 1 (the default), each mapper receives exactly one line of input. The `mapreduce.input.lineinputformat.linespermap` property controls the value of N . By way of example, consider these four lines again:

```
On the top of the Crumpty Tree  
The Quangle Wangle sat,  
But his face you could not see,  
On account of his Beaver Hat.
```

If, for example, N is 2, then each split contains two lines. One mapper will receive the first two key-value pairs:

```
(0, On the top of the Crumpty Tree)  
(33, The Quangle Wangle sat,)
```

And another mapper will receive the second two key-value pairs:

```
(57, But his face you could not see,)  
(89, On account of his Beaver Hat.)
```

The keys and values are the same as those that `TextInputFormat` produces. The difference is in the way the splits are constructed.

Usually, having a map task for a small number of lines of input is inefficient (due to the overhead in task setup), but there are applications that take a small amount of input data and run an extensive (i.e., CPU-intensive) computation for it, then emit their output. Simulations are a good example. By creating an input file that specifies input parameters, one per line, you can perform a *parameter sweep*: run a set of simulations in parallel to find how a model varies as the parameter changes.



If you have long-running simulations, you may fall afoul of task timeouts. When a task doesn't report progress for more than 10 minutes, the application master assumes it has failed and aborts the process (see ["Task Failure" on page 193](#)).

The best way to guard against this is to report progress periodically, by writing a status message or incrementing a counter, for example. See ["What Constitutes Progress in MapReduce?" on page 191](#).

Another example is using Hadoop to bootstrap data loading from multiple data sources, such as databases. You create a "seed" input file that lists the data sources, one per line. Then each mapper is allocated a single data source, and it loads the data from that source into HDFS. The job doesn't need the reduce phase, so the number of reducers should be set to zero (by calling `setNumReduceTasks()` on `Job`). Furthermore, MapReduce jobs can be run to process the data loaded into HDFS. See [Appendix C](#) for an example.

XML

Most XML parsers operate on whole XML documents, so if a large XML document is made up of multiple input splits, it is a challenge to parse these individually. Of course, you can process the entire XML document in one mapper (if it is not too large) using the technique in ["Processing a whole file as a record" on page 228](#).

Large XML documents that are composed of a series of "records" (XML document fragments) can be broken into these records using simple string or regular-expression matching to find the start and end tags of records. This alleviates the problem when the document is split by the framework because the next start tag of a record is easy to find by simply scanning from the start of the split, just like `TextInputFormat` finds newline boundaries.

Hadoop comes with a class for this purpose called `StreamXmlRecordReader` (which is in the `org.apache.hadoop.streaming.mapreduce` package, although it can be used outside of Streaming). You can use it by setting your input format to `StreamInputFormat` and setting the `stream.recordreader.class` property to `org.apache.hadoop.streaming.mapreduce.StreamXmlRecordReader`. The reader is configured by setting job configuration properties to tell it the patterns for the start and end tags (see the class documentation for details).⁴

To take an example, Wikipedia provides dumps of its content in XML form, which are appropriate for processing in parallel with MapReduce using this approach. The data is contained in one large XML wrapper document, which contains a series of elements,

4. See [Mahout's XmlInputFormat](#) for an improved XML input format.

such as page elements that contain a page’s content and associated metadata. Using `StreamXmlRecordReader`, the page elements can be interpreted as records for processing by a mapper.

Binary Input

Hadoop MapReduce is not restricted to processing textual data. It has support for binary formats, too.

SequenceFileInputFormat

Hadoop’s sequence file format stores sequences of binary key-value pairs. Sequence files are well suited as a format for MapReduce data because they are splittable (they have sync points so that readers can synchronize with record boundaries from an arbitrary point in the file, such as the start of a split), they support compression as a part of the format, and they can store arbitrary types using a variety of serialization frameworks. (These topics are covered in “[SequenceFile](#)” on page 127.)

To use data from sequence files as the input to MapReduce, you can use `SequenceFileInputFormat`. The keys and values are determined by the sequence file, and you need to make sure that your map input types correspond. For example, if your sequence file has `IntWritable` keys and `Text` values, like the one created in [Chapter 5](#), then the map signature would be `Mapper<IntWritable, Text, K, V>`, where `K` and `V` are the types of the map’s output keys and values.



Although its name doesn’t give it away, `SequenceFileInputFormat` can read map files as well as sequence files. If it finds a directory where it was expecting a sequence file, `SequenceFileInputFormat` assumes that it is reading a map file and uses its datafile. This is why there is no `MapFileInputFormat` class.

SequenceFileAsTextInputFormat

`SequenceFileAsTextInputFormat` is a variant of `SequenceFileInputFormat` that converts the sequence file’s keys and values to `Text` objects. The conversion is performed by calling `toString()` on the keys and values. This format makes sequence files suitable input for Streaming.

SequenceFileAsBinaryInputFormat

`SequenceFileAsBinaryInputFormat` is a variant of `SequenceFileInputFormat` that retrieves the sequence file’s keys and values as opaque binary objects. They are encapsulated as `BytesWritable` objects, and the application is free to interpret the underlying byte array as it pleases. In combination with a process that creates sequence files with `SequenceFile.Writer`’s `appendRaw()` method or

`SequenceFileAsBinaryOutputFormat`, this provides a way to use any binary data types with MapReduce (packaged as a sequence file), although plugging into Hadoop's serialization mechanism is normally a cleaner alternative (see “[Serialization Frameworks](#)” on page 126).

FixedLengthInputFormat

`FixedLengthInputFormat` is for reading fixed-width binary records from a file, when the records are not separated by delimiters. The record size must be set via `fixedLengthInputFormat.record.length`.

Multiple Inputs

Although the input to a MapReduce job may consist of multiple input files (constructed by a combination of file globs, filters, and plain paths), all of the input is interpreted by a single `InputFormat` and a single `Mapper`. What often happens, however, is that the data format evolves over time, so you have to write your mapper to cope with all of your legacy formats. Or you may have data sources that provide the same type of data but in different formats. This arises in the case of performing joins of different datasets; see “[Reduce-Side Joins](#)” on page 270. For instance, one might be tab-separated plain text, and the other a binary sequence file. Even if they are in the same format, they may have different representations, and therefore need to be parsed differently.

These cases are handled elegantly by using the `MultipleInputs` class, which allows you to specify which `InputFormat` and `Mapper` to use on a per-path basis. For example, if we had weather data from the UK Met Office⁵ that we wanted to combine with the NCDC data for our maximum temperature analysis, we might set up the input as follows:

```
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

This code replaces the usual calls to `FileInputFormat.addInputPath()` and `job.setMapperClass()`. Both the Met Office and NCDC data are text based, so we use `TextInputFormat` for each. But the line format of the two data sources is different, so we use two different mappers. The `MaxTemperatureMapper` reads NCDC input data and extracts the year and temperature fields. The `MetOfficeMaxTemperatureMapper` reads Met Office input data and extracts the year and temperature fields. The important thing is that the map outputs have the same types, since the reducers (which are all of the same type) see the aggregated map outputs and are not aware of the different mappers used to produce them.

5. Met Office data is generally available only to the research and academic community. However, there is a small amount of monthly weather station data available at <http://www.metoffice.gov.uk/climate/uk/stationdata/>.

The `MultipleInputs` class has an overloaded version of `addInputPath()` that doesn't take a mapper:

```
public static void addInputPath(Job job, Path path,  
                                Class<? extends InputFormat> inputFormatClass)
```

This is useful when you only have one mapper (set using the Job's `setMapperClass()` method) but multiple input formats.

Database Input (and Output)

`DBInputFormat` is an input format for reading data from a relational database, using JDBC. Because it doesn't have any sharding capabilities, you need to be careful not to overwhelm the database from which you are reading by running too many mappers. For this reason, it is best used for loading relatively small datasets, perhaps for joining with larger datasets from HDFS using `MultipleInputs`. The corresponding output format is `DBOutputFormat`, which is useful for dumping job outputs (of modest size) into a database.

For an alternative way of moving data between relational databases and HDFS, consider using Sqoop, which is described in [Chapter 15](#).

HBase's `TableInputFormat` is designed to allow a MapReduce program to operate on data stored in an HBase table. `TableOutputFormat` is for writing MapReduce outputs into an HBase table.

Output Formats

Hadoop has output data formats that correspond to the input formats covered in the previous section. The `OutputFormat` class hierarchy appears in [Figure 8-4](#).

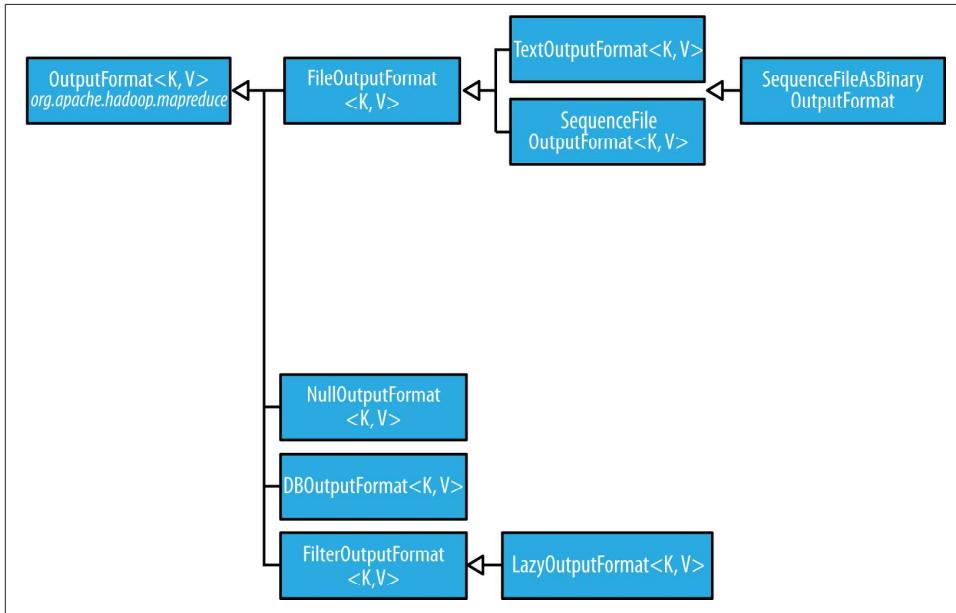


Figure 8-4. OutputFormat class hierarchy

Text Output

The default output format, `TextOutputFormat`, writes records as lines of text. Its keys and values may be of any type, since `TextOutputFormat` turns them to strings by calling `toString()` on them. Each key-value pair is separated by a tab character, although that may be changed using the `mapreduce.output.textoutputformat.separator` property. The counterpart to `TextOutputFormat` for reading in this case is `KeyValueTextInputFormat`, since it breaks lines into key-value pairs based on a configurable separator (see “[KeyValueTextInputFormat](#)” on page 233).

You can suppress the key or the value from the output (or both, making this output format equivalent to `NullOutputFormat`, which emits nothing) using a `NullWritable` type. This also causes no separator to be written, which makes the output suitable for reading in using `TextInputFormat`.

Binary Output

SequenceFileOutputFormat

As the name indicates, `SequenceFileOutputFormat` writes sequence files for its output. This is a good choice of output if it forms the input to a further MapReduce job, since it is compact and is readily compressed. Compression is controlled via the static methods on `SequenceFileOutputFormat`, as described in “[Using Compression in MapReduce](#)”

on page 107. For an example of how to use `SequenceFileOutputFormat`, see “[Sorting](#)” on page 255.

SequenceFileAsBinaryOutputFormat

`SequenceFileAsBinaryOutputFormat`—the counterpart to `SequenceFileAsBinaryInputFormat`—writes keys and values in raw binary format into a sequence file container.

MapFileOutputFormat

`MapFileOutputFormat` writes map files as output. The keys in a MapFile must be added in order, so you need to ensure that your reducers emit keys in sorted order.



The reduce *input* keys are guaranteed to be sorted, but the output keys are under the control of the reduce function, and there is nothing in the general MapReduce contract that states that the reduce *output* keys have to be ordered in any way. The extra constraint of sorted reduce output keys is just needed for `MapFileOutputFormat`.

Multiple Outputs

`FileOutputFormat` and its subclasses generate a set of files in the output directory. There is one file per reducer, and files are named by the partition number: *part-r-00000*, *part-r-00001*, and so on. Sometimes there is a need to have more control over the naming of the files or to produce multiple files per reducer. MapReduce comes with the `MultipleOutputs` class to help you do this.⁶

An example: Partitioning data

Consider the problem of partitioning the weather dataset by weather station. We would like to run a job whose output is one file per station, with each file containing all the records for that station.

One way of doing this is to have a reducer for each weather station. To arrange this, we need to do two things. First, write a partitioner that puts records from the same weather station into the same partition. Second, set the number of reducers on the job to be the number of weather stations. The partitioner would look like this:

6. The old MapReduce API includes two classes for producing multiple outputs: `MultipleOutputFormat` and `MultipleOutputs`. In a nutshell, `MultipleOutputs` is more fully featured, but `MultipleOutputFormat` has more control over the output directory structure and file naming. `MultipleOutputs` in the new API combines the best features of the two multiple output classes in the old API. The code on this book’s website includes old API equivalents of the examples in this section using both `MultipleOutputs` and `MultipleOutputFormat`.

```

public class StationPartitioner extends Partitioner<LongWritable, Text> {

    private NcdcRecordParser parser = new NcdcRecordParser();

    @Override
    public int getPartition(LongWritable key, Text value, int numPartitions) {
        parser.parse(value);
        return getPartition(parser.getStationId());
    }

    private int getPartition(String stationId) {
        ...
    }

}

```

The `getPartition(String)` method, whose implementation is not shown, turns the station ID into a partition index. To do this, it needs a list of all the station IDs; it then just returns the index of the station ID in the list.

There are two drawbacks to this approach. The first is that since the number of partitions needs to be known before the job is run, so does the number of weather stations. Although the NCDC provides metadata about its stations, there is no guarantee that the IDs encountered in the data will match those in the metadata. A station that appears in the metadata but not in the data wastes a reduce task. Worse, a station that appears in the data but not in the metadata doesn't get a reduce task; it has to be thrown away. One way of mitigating this problem would be to write a job to extract the unique station IDs, but it's a shame that we need an extra job to do this.

The second drawback is more subtle. It is generally a bad idea to allow the number of partitions to be rigidly fixed by the application, since this can lead to small or uneven-sized partitions. Having many reducers doing a small amount of work isn't an efficient way of organizing a job; it's much better to get reducers to do more work and have fewer of them, as the overhead in running a task is then reduced. Uneven-sized partitions can be difficult to avoid, too. Different weather stations will have gathered a widely varying amount of data; for example, compare a station that opened one year ago to one that has been gathering data for a century. If a few reduce tasks take significantly longer than the others, they will dominate the job execution time and cause it to be longer than it needs to be.



There are two special cases when it does make sense to allow the application to set the number of partitions (or equivalently, the number of reducers):

Zero reducers

This is a vacuous case: there are no partitions, as the application needs to run only map tasks.

One reducer

It can be convenient to run small jobs to combine the output of previous jobs into a single file. This should be attempted only when the amount of data is small enough to be processed comfortably by one reducer.

It is much better to let the cluster drive the number of partitions for a job, the idea being that the more cluster resources there are available, the faster the job can complete. This is why the default `HashPartitioner` works so well: it works with any number of partitions and ensures each partition has a good mix of keys, leading to more evenly sized partitions.

If we go back to using `HashPartitioner`, each partition will contain multiple stations, so to create a file per station, we need to arrange for each reducer to write multiple files. This is where `MultipleOutputs` comes in.

MultipleOutputs

`MultipleOutputs` allows you to write data to files whose names are derived from the output keys and values, or in fact from an arbitrary string. This allows each reducer (or mapper in a map-only job) to create more than a single file. Filenames are of the form `name-m-nnnnn` for map outputs and `name-r-nnnnn` for reduce outputs, where `name` is an arbitrary name that is set by the program and `nnnnn` is an integer designating the part number, starting from `00000`. The part number ensures that outputs written from different partitions (mappers or reducers) do not collide in the case of the same name.

The program in [Example 8-5](#) shows how to use `MultipleOutputs` to partition the dataset by station.

Example 8-5. Partitioning whole dataset into files named by the station ID using `MultipleOutputs`

```
public class PartitionByStationUsingMultipleOutputs extends Configured
    implements Tool {

    static class StationMapper
        extends Mapper<LongWritable, Text, Text, Text> {

        private NcdcRecordParser parser = new NcdcRecordParser();
```

```

@Override
protected void map(LongWritable key, Text value, Context context)
    throws IOException, InterruptedException {
    parser.parse(value);
    context.write(new Text(parser.getStationId()), value);
}
}

static class MultipleOutputsReducer
    extends Reducer<Text, Text, NullWritable, Text> {

    private MultipleOutputs<NullWritable, Text> multipleOutputs;

    @Override
    protected void setup(Context context)
        throws IOException, InterruptedException {
        multipleOutputs = new MultipleOutputs<NullWritable, Text>(context);
    }

    @Override
    protected void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {
        for (Text value : values) {
            multipleOutputs.write(NullWritable.get(), value, key.toString());
        }
    }

    @Override
    protected void cleanup(Context context)
        throws IOException, InterruptedException {
        multipleOutputs.close();
    }
}

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

    job.setMapperClass(StationMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setReducerClass(MultipleOutputsReducer.class);
    job.setOutputKeyClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new PartitionByStationUsingMultipleOutputs(),
        args);
    System.exit(exitCode);
}

```

```
    }  
}
```

In the reducer, which is where we generate the output, we construct an instance of `MultipleOutputs` in the `setup()` method and assign it to an instance variable. We then use the `MultipleOutputs` instance in the `reduce()` method to write to the output, in place of the context. The `write()` method takes the key and value, as well as a name. We use the station identifier for the name, so the overall effect is to produce output files with the naming scheme `station_identifier-r-nnnnn`.

In one run, the first few output files were named as follows:

```
output/010010-99999-r-00027  
output/010050-99999-r-00013  
output/010100-99999-r-00015  
output/010280-99999-r-00014  
output/010550-99999-r-00000  
output/010980-99999-r-00011  
output/011060-99999-r-00025  
output/012030-99999-r-00029  
output/012350-99999-r-00018  
output/012620-99999-r-00004
```

The base path specified in the `write()` method of `MultipleOutputs` is interpreted relative to the output directory, and because it may contain file path separator characters (/), it's possible to create subdirectories of arbitrary depth. For example, the following modification partitions the data by station and year so that each year's data is contained in a directory named by the station ID (such as `029070-99999/1901/part-r-00000`):

```
@Override  
protected void reduce(Text key, Iterable<Text> values, Context context)  
    throws IOException, InterruptedException {  
    for (Text value : values) {  
        parser.parse(value);  
        String basePath = String.format("%s/%s/part",  
            parser.getStationId(), parser.getYear());  
        multipleOutputs.write(NullWritable.get(), value, basePath);  
    }  
}
```

`MultipleOutputs` delegates to the mapper's `OutputFormat`. In this example it's a `TextOutputFormat`, but more complex setups are possible. For example, you can create named outputs, each with its own `OutputFormat` and key and value types (which may differ from the output types of the mapper or reducer). Furthermore, the mapper or reducer (or both) may write to multiple output files for each record processed. Consult the Java documentation for more information.

Lazy Output

`FileOutputFormat` subclasses will create output (*part-r-nnnnn*) files, even if they are empty. Some applications prefer that empty files not be created, which is where `LazyOutputFormat` helps. It is a wrapper output format that ensures that the output file is created only when the first record is emitted for a given partition. To use it, call its `setOutputFormatClass()` method with the `JobConf` and the underlying output format.

Streaming supports a `-lazyOutput` option to enable `LazyOutputFormat`.

Database Output

The output formats for writing to relational databases and to HBase are mentioned in “[Database Input \(and Output\)](#)” on page 238.