Reading and writing to HDFS programmatically

To motivate an examination of the HDFS Java API, we'll develop a PutMerge program for merging files while putting them into HDFS. The command line utilities don't support this operation; we'll use the API.

The motivation for this example came when we wanted to analyze Apache log files coming from many web servers. We can copy each log file into HDFS, but in general, Hadoop works more effectively with a single large file rather than a number of smaller ones. ("Smaller" is relative here as it can still be tens or hundreds of gigabytes.) Besides, for analytics purposes we think of the log data as one big file. That it's spread over multiple files is an incidental result of the physical web server architecture. One solution is to merge all the files first and then copy the combined file into HDFS. Unfortunately, the file merging will require a lot of disk space in the local machine. It would be much easier if we could merge all the files on the fly as we copy them into HDFS.

What we need is, therefore, a PutMerge-type of operation. Hadoop's command line utilities include a `getmerge` command for merging a number of HDFS files before copying them onto the local machine. What we're looking for is the exact opposite. This is not available in Hadoop's file utilities. We'll write our own program using the HDFS API.

The main classes for file manipulation in Hadoop are in the package org.apache.hadoop.fs. Basic Hadoop file operations include the familiar open, read, write, and close. In fact, the Hadoop file API is generic and can be used for working with filesystems other than HDFS. For our PutMerge program, we'll use the Hadoop file API to both read the local filesystem and write to HDFS.

The starting point for the Hadoop file API is the FileSystem class. This is an abstract class for interfacing with the filesystem, and there are different concrete subclasses for handling HDFS and the local filesystem. You get the desired `FileSystem` instance by calling the factory method `FileSystem.get(Configuration conf)`. The `Configuration` class is a special class for holding key/value configuration parameters. Its default instantiation is based on the resource configuration for your HDFS system. We can get the `FileSystem` object to interface with HDFS by

```
Configuration conf = new Configuration();
FileSystem hdfs = FileSystem.get(conf);
```

To get a `FileSystem` object specifically for the local filesystem, there's the `FileSystem.getLocal(Configuration conf)` factory method.

```
FileSystem local = FileSystem.getLocal(conf);
```

Hadoop file API uses `Path` objects to encode file and directory names and `FileStatus` objects to store metadata for files and directories. Our PutMerge program will merge all files from a local directory. We use the `FileSystem`'s `listStatus()` method to get a list of files in a directory.

```
Path inputDir = new Path(args[0]);
FileStatus[] inputFiles = local.listStatus(inputDir);
```

The length of the `inputFiles` array is the number of files in the specified directory. Each `FileStatus` object in `inputFiles` has metadata information such as file length, permissions, modification time, and others. Of interest to our PutMerge program is each file's `Path` representation, `inputFiles[i].getPath()`. We can use this `Path` to request an `FSDataInputStream` object for reading in the file.

```
FSDataInputStream in = local.open(inputFiles[i].getPath());
byte buffer[] = new byte[256];
int bytesRead = 0;
while( (bytesRead = in.read(buffer)) > 0) {
    ...
}
in.close();
```

`FSDataInputStream` is a subclass of Java's standard java.io.DataInputStream with additional support for random access. For writing to a HDFS file, there's the analogous `FSDataOutputStream` object.

```
Path hdfsFile = new Path(args[1]);
FSDataOutputStream out = hdfs.create(hdfsFile);
out.write(buffer, 0, bytesRead);
out.close();
```

To complete the PutMerge program, we create a loop that goes through all the files in `inputFiles` as we read each one in and write it out to the destination HDFS file.

```
import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class PutMerge {

  public static void main(String[] args) throws IOException {

      Configuration conf = new Configuration();
      FileSystem hdfs  = FileSystem.get(conf);
      FileSystem local = FileSystem.getLocal(conf);

      Path inputDir = new Path(args[0]);
      Path hdfsFile = new Path(args[1]);

      try {
          FileStatus[] inputFiles = local.listStatus(inputDir);
          FSDataOutputStream out = hdfs.create(hdfsFile);

          for (int i=0; i<inputFiles.length; i++) {
              System.out.println(inputFiles[i].getPath().getName());
              FSDataInputStream in =
      local.open(inputFiles[i].getPath());
              byte buffer[] = new byte[256];
              int bytesRead = 0;
              while( (bytesRead = in.read(buffer)) > 0) {
                  out.write(buffer, 0, bytesRead);
              }
              in.close();
          }
          out.close();
      } catch (IOException e) {
          e.printStackTrace();
      }
  }
```

1 Specify input directory and output file

2 Get list of local files

3 Create HDFS output stream

4 Open local input stream

The general flow of the program involves first setting the local directory and the HDFS destination file based on user-specified arguments ❶. In ❷ we extract information about each file in the local input directory. We create an output stream to write to the HDFS file in ❸. We loop through each file in the local directory, and ❹ opens an input stream to read that file. The rest of the code is standard Java file copy.

The `FileSystem` class also has methods such as `delete()`, `exists()`, `mkdirs()`, and `rename()` for other standard file operations. You can find the most recent Javadoc for the Hadoop file API at http://hadoop.apache.org/core/docs/current/api/org/apache/hadoop/fs/package-summary.html.

We have covered how to work with files in HDFS. You now know a few ways to put data into and out of HDFS. But merely having data isn't terribly interesting. You want to process it, analyze it, and do other things. Let's conclude our discussion of HDFS and move on to the other major component of Hadoop, the MapReduce framework, and how to program under it.

Anatomy of a MapReduce program

As we have mentioned before, a MapReduce program processes data by manipulating (key/value) pairs in the general form

map: (K1,V1) → list(K2,V2)

reduce: (K2,list(V2)) → list(K3,V3)

Not surprisingly, this is an overly generic representation of the data flow. In this section we learn more details about each stage in a typical MapReduce program. Figure 3.1 displays a high-level diagram of the entire process, and we further dissect each component as we step through the flow.

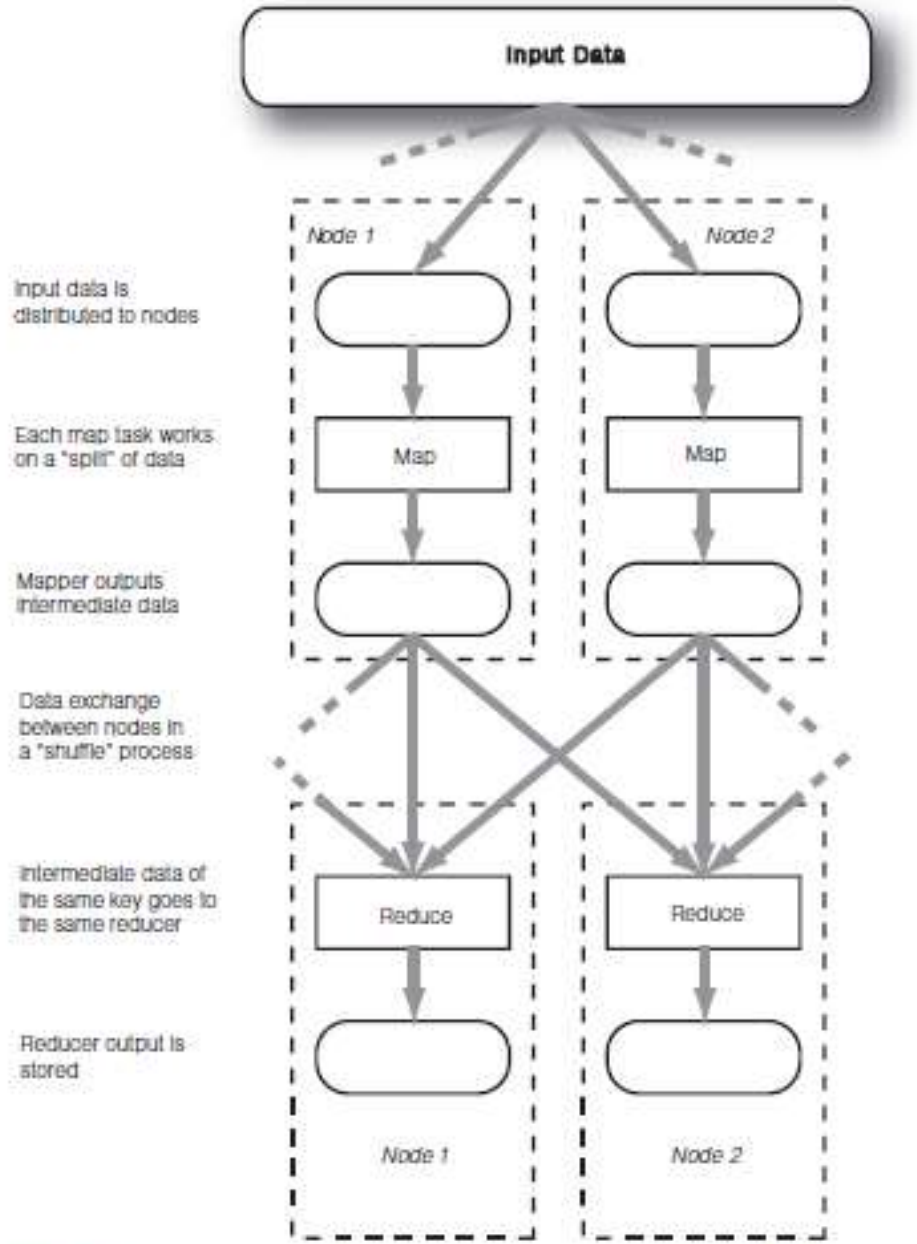


Figure 3.1 The general MapReduce data flow. Note that after distributing input data to different nodes, the only time nodes communicate with each other is at the "shuffle" step. This restriction on communication greatly helps scalability.

Hadoop data types

Despite our many discussions regarding keys and values, we have yet to mention their types. The MapReduce framework won't allow them to be any arbitrary class. For example, although we can and often do talk about certain keys and values as integers, strings, and so on, they aren't exactly standard Java classes, such as Integer, String, and so forth. This is because the MapReduce framework has a certain defined way of serializing the key/value pairs to move them across the cluster's network, and only classes that support this kind of serialization can function as keys or values in the framework.

More specifically, classes that implement the Writable interface can be values, and classes that implement the `WritableComparable<T>` interface can be either keys or values. Note that the `WritableComparable<T>` interface is a combination of the `Writable` and `java.lang.Comparable<T>` interfaces. We need the comparability requirement for keys because they will be sorted at the reduce stage, whereas values are simply passed through.

Hadoop comes with a number of predefined classes that implement `WritableComparable`, including wrapper classes for all the basic data types, as seen in table 3.1.

Table 3.1 List of frequently used types for the key/value pairs. These classes all implement the `WritableComparable` interface.

Class	Description
`BooleanWritable`	Wrapper for a standard Boolean variable
`ByteWritable`	Wrapper for a single byte
`DoubleWritable`	Wrapper for a Double
`FloatWritable`	Wrapper for a Float
`IntWritable`	Wrapper for a Integer
`LongWritable`	Wrapper for a Long
`Text`	Wrapper to store text using the UTF8 format
`NullWritable`	Placeholder when the key or value is not needed

Keys and values can take on types beyond the basic ones which Hadoop natively supports. You can create your own custom type as long as it implements the `Writable` (or `WritableComparable<T>`) interface. For example, listing 3.2 shows a class that can represent edges in a network. This may represent a flight route between two cities.

Listing 3.2 An example class that implements the WritableComparable interface

```
public class Edge implements WritableComparable<Edge>{

    private String departureNode;
    private String arrivalNode;

    public String getDepartureNode() { return departureNode;}

    @Override
    public void readFields(DataInput in) throws IOException {
        departureNode = in.readUTF();
        arrivalNode = in.readUTF();
    }

    @Override
    public void write(DataOutput out) throws IOException {
        out.writeUTF(departureNode);
        out.writeUTF(arrivalNode);
    }

    @Override
    public int compareTo(Edge o) {
        return (departureNode.compareTo(o.departureNode) != 0)
            ? departureNode.compareTo(o.departureNode)
            : arrivalNode.compareTo(o.arrivalNode);
    }
}
```

1 Specify how to read data in

2 Specify how to write data out

3 Define ordering of data

The `Edge` class implements the `readFields()` 1 and `write()` 2 methods of the `Writable` interface. They work with the Java `DataInput` and `DataOutput` classes to serialize the class contents. Implement the `compareTo()` method 3 for the `Comparable` interface. It returns -1, 0, or 1 if the called `Edge` is less than, equal to, or greater than the given `Edge`.

Mapper

To serve as the mapper, a class implements from the Mapper interface and inherits the `MapReduceBase` class. The `MapReduceBase` class, not surprisingly, serves as the base class for both mappers and reducers. It includes two methods that effectively act as the constructor and destructor for the class:

- *void configure(JobConf job)*—In this function you can extract the parameters set either by the configuration XML files or in the main class of your application. Call this function before any data processing begins.
- *void close()*—As the last action before the map task terminates, this function should wrap up any loose ends—database connections, open files, and so on.

The `Mapper` interface is responsible for the data processing step. It utilizes Java generics of the form `Mapper<K1,V1,K2,V2>` where the key classes and value classes implement the `WritableComparable` and `Writable` interfaces, respectively. Its single method is to process an individual (key/value) pair:

```
void map(K1 key,
         V1 value,
         OutputCollector<K2,V2> output,
         Reporter reporter
        ) throws IOException
```

The function generates a (possibly empty) list of (K2, V2) pairs for a given (K1, V1) input pair. The `OutputCollector` receives the output of the mapping process, and the `Reporter` provides the option to record extra information about the mapper as the task progresses.

Hadoop provides a few useful mapper implementations. You can see some of them in the table 3.2.

Table 3.2 Some useful `Mapper` implementations predefined by Hadoop

Class	Description
`IdentityMapper<K,V>`	Implements Mapper<K,V,K,V> and maps inputs directly to outputs
`InverseMapper<K,V>`	Implements Mapper<K,V,V,K> and reverses the key/value pair
`RegexMapper<K>`	Implements Mapper<K,Text,Text,LongWritable> and generates a (match, 1) pair for every regular expression match
`TokenCountMapper<K>`	Implements Mapper<K,Text,Text,LongWritable> and generates a (token, 1) pair when the input value is tokenized

As the MapReduce name implies, the major data flow operation after map is the reduce phase

Reducer

As with any mapper implementation, a reducer must first extend the MapReduce base class to allow for configuration and cleanup. In addition, it must also implement the `Reducer` interface which has the following single method:

```
void reduce(K2 key,
            Iterator<V2> values,
            OutputCollector<K3,V3> output,
            Reporter reporter
           ) throws IOException
```

When the reducer task receives the output from the various mappers, it sorts the incoming data on the key of the (key/value) pair and groups together all values of the same key. The `reduce()` function is then called, and it generates a (possibly empty) list of (K3, V3) pairs by iterating over the values associated with a given key. The `OutputCollector` receives the output of the reduce process and writes it to an output file. The `Reporter` provides the option to record extra information about the reducer as the task progresses.

Table 3.3 Some useful `Reducer` implementations predefined by Hadoop

Class	Description
`IdentityReducer<K,V>`	Implements Reducer<K,V,K,V> and maps inputs directly to outputs
`LongSumReducer<K>`	Implements Reducer<K,LongWritable,K,LongWritable> and determines the sum of all values corresponding to the given key

Although we have referred to Hadoop programs as MapReduce applications, there is a vital step between the two stages: directing the result of the mappers to the different reducers. This is the responsibility of the partitioner.

Partitioner—redirecting output from Mapper

A common misconception for first-time MapReduce programmers is to use only a single reducer. After all, a single reducer sorts all of your data before processing—and who doesn't like sorted data? Our discussions regarding MapReduce expose the folly of such thinking. We would have ignored the benefits of parallel computation. With one reducer, our compute cloud has been demoted to a compute raindrop.

With multiple reducers, we need some way to determine the appropriate one to send a (key/value) pair outputted by a mapper. The default behavior is to hash the key to determine the reducer. Hadoop enforces this strategy by use of the `HashPartitioner` class. Sometimes the `HashPartitioner` will steer you awry. Let's return to the `Edge` class introduced in section 3.2.1.

Suppose you used the `Edge` class to analyze flight information data to determine the number of passengers departing from each airport. Such data may be

(San Francisco, Los Angeles)	Chuck Lam
(San Francisco, Dallas)	James Warren
...	

If you used `HashPartitioner`, the two rows could be sent to different reducers. The number of departures would be processed twice and both times erroneously.

How do we customize the partitioner for your applications? In this situation, we want all edges with a common departure point to be sent to the same reducer. This is done easily enough by hashing the `departureNode` member of the `Edge`:

```
public class EdgePartitioner implements Partitioner<Edge, Writable>
{
    @Override
    public int getPartition(Edge key, Writable value, int numPartitions)
    {
        return key.getDepartureNode().hashCode() % numPartitions;
    }
    @Override
    public void configure(JobConf conf) { }
}
```

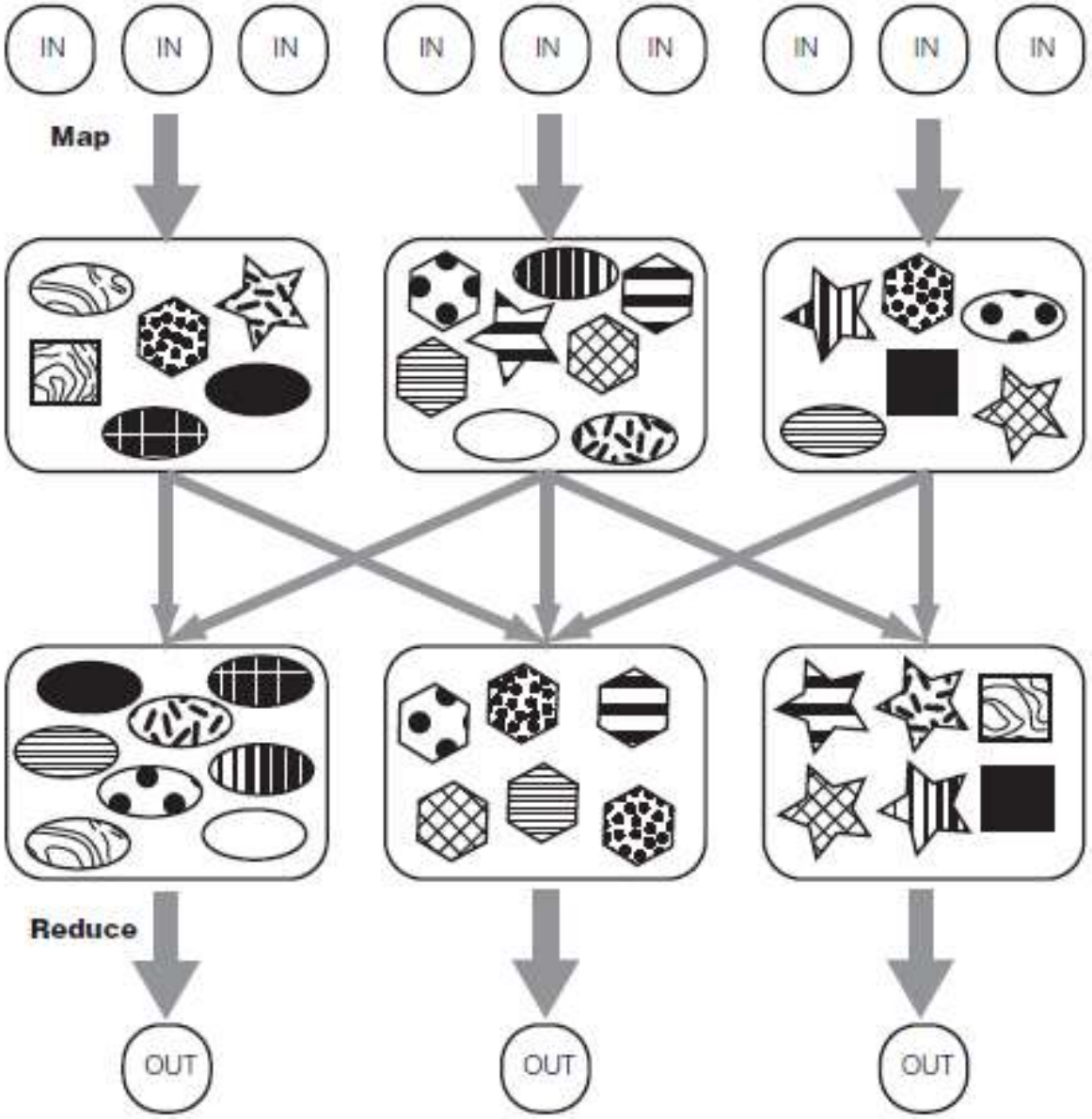
A custom partitioner only needs to implement two functions: `configure()` and `getPartition()`. The former uses the Hadoop job configuration to configure the partitioner, and the latter returns an integer between 0 and the number of reduce tasks indexing to which reducer the (key/value) pair will be sent.

The exact mechanics of the partitioner may be difficult to follow. Figure 3.2 illustrates this for better understanding.

Between the map and reduce stages, a MapReduce application must take the output from the mapper tasks and distribute the results among the reducer tasks. This process is typically called *shuffling*, because the output of a mapper on a single node may be sent to reducers across multiple nodes in the cluster.

Combiner—local reduce

In many situations with MapReduce applications, we may wish to perform a "local reduce" before we distribute the mapper results. Consider the `WordCount` example of



Word counting with predefined mapper and reducer classes

We have concluded our preliminary coverage of all the basic components of MapReduce. Now that you've seen more classes provided by Hadoop, it'll be fun to revisit the `WordCount` example (see listing 3.3), using some of the classes we've learned.

Listing 3.3 Revised version of the `WordCount` example

```
public class WordCount2 {
    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(WordCount2.class);

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

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(LongWritable.class);
        conf.setMapperClass(TokenCountMapper.class);
        conf.setCombinerClass(LongSumReducer.class);
        conf.setReducerClass(LongSumReducer.class);

        client.setConf(conf);
        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

1 Hadoop's own TokenCountMapper

2 Hadoop's own LongSumReducer

We have to write only the driver for this MapReduce program because we have used Hadoop's predefined `TokenCountMapper` class 1 and `LongSumReducer` class 2. Easy, isn't it? Hadoop provides the ability to generate more sophisticated programs (this will be the focus of part 2 of the book), but we want to emphasize that Hadoop allows you to rapidly generate useful programs with a minimal amount of code.

Reading and writing

Let's see how MapReduce reads input data and writes output data and focus on the file formats it uses. To enable easy distributed processing, MapReduce makes certain assumptions about the data it's processing. It also provides flexibility in dealing with a variety of data formats.

Input data usually resides in large files, typically tens or hundreds of gigabytes or even more. One of the fundamental principles of MapReduce's processing power is the splitting of the input data into *chunks*. You can process these chunks in parallel using multiple machines. In Hadoop terminology these chunks are called *input splits*.

The size of each split should be small enough for a more granular parallelization. (If all the input data is in one split, then there is no parallelization.) On the other hand, each split shouldn't be so small that the overhead of starting and stopping the processing of a split becomes a large fraction of execution time.

The principle of dividing input data (which often can be one single massive file) into splits for parallel processing explains some of the design decisions behind Hadoop's generic `FileSystem` as well as HDFS in particular. For example, Hadoop's `FileSystem` provides the class `FSDataInputStream` for file reading rather than using Java's `java.io.DataInputStream`. `FSDataInputStream` extends `DataInputStream` with random read access, a feature that MapReduce requires because a machine may be assigned to process a split that sits right in the middle of an input file. Without random access, it would be extremely inefficient to have to read the file from the beginning until you reach the location of the split. You can also see how HDFS is designed for storing data that MapReduce will split and process in parallel. HDFS stores files in blocks spread over multiple machines. Roughly speaking, each file block is a split. As different machines will likely have different blocks, parallelization is automatic if each split/block is processed by the machine that it's residing at. Furthermore, as HDFS replicates blocks in multiple nodes for reliability, MapReduce can choose any of the nodes that have a copy of a split/block.

Input splits and record boundaries

Note that input splits are a logical division of your records whereas HDFS blocks are a physical division of the input data. It's extremely efficient when they're the same but in practice it's never perfectly aligned. Records may cross block boundaries. Hadoop guarantees the processing of all records. A machine processing a particular split may fetch a fragment of a record from a block other than its "main" block and which may reside remotely. The communication cost for fetching a record fragment is inconsequential because it happens relatively rarely.

You'll recall that MapReduce works on key/value pairs. So far we've seen that Hadoop by default considers each line in the input file to be a record and the key/value pair is the byte offset (key) and content of the line (value), respectively. You may not have recorded all your data that way. Hadoop supports a few other data formats and allows you to define your own.

InputFormat

The way an input file is split up and read by Hadoop is defined by one of the implementations of the `InputFormat` interface. `TextInputFormat` is the default `InputFormat` implementation, and it's the data format we've been implicitly using up to now. It's often useful for input data that has no definite key value, when you want to get the content one line at a time. The key returned by `TextInputFormat` is the byte offset of each line, and we have yet to see any program that uses that key for its data processing.

Table 3.4 Main `InputFormat` classes. `TextInputFormat` is the default unless an alternative is specified. The object type for key and value are also described.

InputFormat	Description
`TextInputFormat`	Each line in the text files is a record. Key is the byte offset of the line, and value is the content of the line. key: LongWritable value: Text
`KeyValueTextInputFormat`	Each line in the text files is a record. The first separator character divides each line. Everything before the separator is the key, and everything after is the value. The separator is set by the key.value.separator.in.input.line property, and the default is the tab (\t) character. key: Text value: Text
`SequenceFileInputFormat<K,V>`	An InputFormat for reading in *sequence files*. Key and value are user defined. Sequence file is a Hadoop-specific compressed binary file format. It's optimized for passing data between the output of one MapReduce job to the input of some other MapReduce job. key: K (user defined) value: V (user defined)
`NLineInputFormat`	Same as TextInputFormat, but each split is guaranteed to have exactly *N* lines. The mapred.line.input.format.linespermap property, which defaults to one, sets *N*. key: LongWritable value: Text

`KeyValueTextInputFormat` is used in the more structured input files where a predefined character, usually a tab (\t), separates the key and value of each line (record). For example, you may have a tab-separated data file of timestamps and URLs:

```
17:16:18    http://hadoop.apache.org/core/docs/r0.19.0/api/index.html
17:16:19    http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html
17:16:20    http://wiki.apache.org/hadoop/GettingStartedWithHadoop
17:16:20    http://www.maxim.com/hotties/2008/finalist_gallery.aspx
17:16:25    http://wiki.apache.org/hadoop/
```

...

You can set your `JobConf` object to use the `KeyValueTextInputFormat` class to read this file.

```
conf.setInputFormat(KeyValueTextInputFormat.class);
```

Given the preceding example file, the first record your mapper reads will have a key of "17:16:18" and a value of "http://hadoop.apache.org/core/docs/r0.19.0/api/index.html". The second record to your mapper will have a key of "17:16:19" and a value of "http://hadoop.apache.org/core/docs/r0.19.0/mapred_tutorial.html." And so on.

Recall that our previous mappers had used `LongWritable` and `Text` as the key and value types, respectively. `LongWritable` is a reasonable type for the key under `TextInputFormat` because the key is a numerical offset. When using `KeyValueTextInputFormat`, both the key and the value will be of type `Text`, and you'll have to change your `Mapper` implementation and `map()` method to reflect the new key type.

CREATING A CUSTOM INPUTFORMAT—INPUTSPLIT AND RECORDREADER

Sometimes you may want to read input data in a way different from the standard `InputFormat` classes. In that case you'll have to write your own custom `InputFormat` class. Let's look at what it involves. `InputFormat` is an interface consisting of only two methods.

```
public interface InputFormat<K, V> {

  InputSplit[] getSplits(JobConf job, int numSplits) throws IOException;

  RecordReader<K, V> getRecordReader(InputSplit split,
                                     JobConf job,
                                     Reporter reporter) throws IOException;
}
```

The two methods sum up the functions that `InputFormat` has to perform:

- Identify all the files used as input data and divide them into input splits. Each map task is assigned one split.
- Provide an object (`RecordReader`) to iterate through records in a given split, and to parse each record into key and value of predefined types.

OutputFormat

MapReduce outputs data into files using the `OutputFormat` class, which is analogous to the `InputFormat` class. The output has no splits, as each reducer writes its output only to its own file. The output files reside in a common directory and are typically named part-*nnnnn*, where *nnnnn* is the partition ID of the reducer. `RecordWriter` objects format the output and `RecordReaders` parse the format of the input.

Hadoop provides several standard implementations of `OutputFormat`, as shown in table 3.5. Not surprisingly, almost all the ones we deal with inherit from the `File OutputFormat` abstract class; `InputFormat` classes inherit from `FileInputFormat`. You specify the `OutputFormat` by calling `setOutputFormat()` of the `JobConf` object that holds the configuration of your MapReduce job.

OutputFormat

MapReduce outputs data into files using the `OutputFormat` class, which is analogous to the `InputFormat` class. The output has no splits, as each reducer writes its output only to its own file. The output files reside in a common directory and are typically named part-*nnnnn*, where *nnnnn* is the partition ID of the reducer. `RecordWriter` objects format the output and `RecordReaders` parse the format of the input.

Hadoop provides several standard implementations of `OutputFormat`, as shown in table 3.5. Not surprisingly, almost all the ones we deal with inherit from the `FileOutputFormat` abstract class; `InputFormat` classes inherit from `FileInputFormat`. You specify the `OutputFormat` by calling `setOutputFormat()` of the `JobConf` object that holds the configuration of your MapReduce job.

Table 3.5 Main `OutputFormat` classes. `TextOutputFormat` is the default.

OutputFormat	Description
`TextOutputFormat<K,V>`	Writes each record as a line of text. Keys and values are written as strings and separated by a tab (\t) character, which can be changed in the mapred.textoutputformat.separator property.
`SequenceFileOutputFormat<K,V>`	Writes the key/value pairs in Hadoop's proprietary sequence file format. Works in conjunction with SequenceFileInputFormat.
`NullOutputFormat<K,V>`	Outputs nothing.

The default `OutputFormat` is `TextOutputFormat`, which writes each record as a line of text. Each record's key and value are converted to strings through `toString()`, and a tab (\t) character separates them. The separator character can be changed in the `mapred.textoutputformat.separator` property.