# Module 4: MapReduce

Previous module | Table of contents | Next module

## Introduction

MapReduce is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. MapReduce programs are written in a particular style influenced by *functional programming* constructs, specifically idioms for processing lists of data. This module explains the nature of this programming model and how it can be used to write programs which run in the Hadoop environment.

## Goals for this Module:

- Understand functional programming as it applies to MapReduce
- Understand the MapReduce program flow
- Understand how to write programs for Hadoop MapReduce
- Learn about additional features of Hadoop designed to aid software development.

## Outline



## Prerequisites

This module requires that you have set up a build environment as described in Module 3. If you have not already configured Hadoop and successfully run the example applications, go back and do so now.

dividing the workload across a large number of machines. This model would not scale to large clusters (hundreds or thousands of nodes) if the components were allowed to share data arbitrarily. The communication overhead required to keep the data on the nodes synchronized at all times would prevent the system from performing reliably or efficiently at large scale.

Instead, all data elements in MapReduce are *immutable*, meaning that they cannot be updated. If in a mapping task you change an input (key, value) pair, it does not get reflected back in the input files; communication occurs only by generating new output (key, value) pairs which are then forwarded by the Hadoop system into the next phase of execution.

#### LIST PROCESSING

Conceptually, MapReduce programs transform lists of input data elements into lists of output data elements. A MapReduce program will do this twice, using two different list processing idioms: *map*, and *reduce*. These terms are taken from several list processing languages such as LISP, Scheme, or ML.

#### MAPPING LISTS

The first phase of a MapReduce program is called *mapping*. A list of data elements are provided, one at a time, to a function called the *Mapper*, which transforms each element individually to an output data element.




Figure 4.1: Mapping creates a new output list by applying a function to individual elements of an input list.

As an example of the utility of map: Suppose you had a function `toUpper(str)` which returns an uppercase version of its input string. You could use this function with `map` to turn a list of strings into a list of uppercase strings. Note that we are not *modifying* the input string here: we are returning a new string that will form part of a new output list.

#### REDUCING LISTS

Reducing lets you aggregate values together. A *reducer* function receives an iterator of input values from an input list. It then combines these values together, returning a single output value.




*Figure 4.2: Reducing a list iterates over the input values to produce an aggregate value as output.*

Reducing is often used to produce "summary" data, turning a large volume of data into a smaller summary of itself. For example, "+" can be used as a reducing function, to return the sum of a list of input values.

#### PUTTING THEM TOGETHER IN MAPREDUCE:

The Hadoop MapReduce framework takes these concepts and uses them to process large volumes of information. A MapReduce program has two components: one that implements the mapper, and another that implements the reducer. The Mapper and Reducer idioms described above are extended slightly to work in this environment, but the basic principles are the same.

**Keys and values:** In MapReduce, no value stands on its own. Every value has a *key* associated with it. Keys identify related values. For example, a log of time-coded speedometer readings from multiple cars could be keyed by license-plate number; it would look like:

```
AAA-123   65mph, 12:00pm
ZZZ-789   50mph, 12:02pm
AAA-123   40mph, 12:05pm
CCC-456   25mph, 12:15pm
...
```

The mapping and reducing functions receive not just values, but (key, value) pairs. The output of each of these functions is the same: both a key and a value must be emitted to the next list in the data flow.

MapReduce is also less strict than other languages about how the Mapper and Reducer work. In more formal

**Keys divide the reduce space:** A reducing function turns a large list of values into one (or a few) output values. In MapReduce, all of the output values are not usually reduced together. All of the values *with the same key* are presented to a single reducer together. This is performed independently of any reduce operations occurring on other lists of values, with different keys attached.




*Figure 4.3: Different colors represent different keys. All values with the same key are presented to a single reduce task.*

### AN EXAMPLE APPLICATION: WORD COUNT

A simple MapReduce program can be written to determine how many times different words appear in a set of files. For example, if we had the files:

**foo.txt:** Sweet, this is the foo file

**bar.txt:** This is the bar file

We would expect the output to be:

```
sweet 1
this  2
is    2
the   2
foo   1
bar   1
file  2
```

Naturally, we can write a program in MapReduce to compute this output. The high-level structure would look like this:

```
mapper (filename, file-contents):
  for each word in file-contents:
    emit (word, 1)

reducer (word, values):
  sum = 0
  for each value in values:
    sum = sum + value
  emit (word, sum)
```

*Listing 4.1: High-Level MapReduce Word Count*

Several instances of the mapper function are created on the different machines in our cluster. Each instance receives a different input file (it is assumed that we have many such files). The mappers output (word, 1) pairs which are then forwarded to the reducers. Several instances of the reducer method are also instantiated on the different machines. Each reducer is responsible for processing the list of values associated with a different word. The list of values will be a list of 1's; the reducer sums up those ones into a final count associated with a single word. The reducer then emits the final (word, count) output which is written to an output file.

We can write a very similar program to this in Hadoop MapReduce; it is included in the Hadoop distribution in `src/examples/org/apache/hadoop/examples/WordCount.java`. It is partially reproduced below:

```
  public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

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

```
 * A reducer class that just emits the sum of the input values.
 */
public static class Reduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {

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

*Listing 4.2: Hadoop MapReduce Word Count Source*

There are some minor differences between this actual Java implementation and the pseudo-code shown above. First, Java has no native `emit` keyword; the `OutputCollector` object you are given as an input will receive values to emit to the next stage of execution. And second, the default input format used by Hadoop presents each line of an input file as a separate input to the mapper function, not the entire file at a time. It also uses a `StringTokenizer` object to break up the line into words. This does not perform any normalization of the input, so "cat", "Cat" and "cat," are all regarded as different strings. Note that the class-variable `word` is reused each time the mapper outputs another (word, 1) pairing; this saves time by not allocating a new variable for each output. The `output.collect()` method will copy the values it receives as input, so you are free to overwrite the variables you use.

#### THE DRIVER METHOD

There is one final component of a Hadoop MapReduce program, called the *Driver*. The driver initializes the job and instructs the Hadoop platform to execute your code on a set of input files, and controls where the output files are placed. A cleaned-up version of the driver from the example Java implementation that comes with Hadoop is presented below:

```
public void run(String inputPath, String outputPath) throws Exception {
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName("wordcount");

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

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

  FileInputFormat.addInputPath(conf, new Path(inputPath));
  FileOutputFormat.setOutputPath(conf, new Path(outputPath));

  JobClient.runJob(conf);
}
```

*Listing 4.3: Hadoop MapReduce Word Count Driver*

This method sets up a job to execute the word count program across all the files in a given input directory (the `inputPath` argument). The output from the reducers are written into files in the directory identified by `outputPath`. The configuration information to run the job is captured in the *JobConf* object. The mapping and reducing functions are identified by the `setMapperClass()` and `setReducerClass()` methods. The data types emitted by the reducer are identified by `setOutputKeyClass()` and `setOutputValueClass()`. By default, it is assumed that these are the output types of the mapper as well. If this is not the case, the methods `setMapOutputKeyClass()` and `setMapOutputValueClass()` methods of the `JobConf` class will override these. The input types fed to the mapper are controlled by the *InputFormat* used. Input formats are discussed in more detail below. The default input format, "TextInputFormat," will load data in as (*LongWritable, Text*) pairs. The long value is the byte offset of the line in the file. The Text object holds the string contents of the line of the file.

The call to `JobClient.runJob(conf)` will submit the job to MapReduce. This call will block until the job completes. If the job fails, it will throw an IOException. JobClient also provides a non-blocking version called

Now that we have seen the components that make up a basic MapReduce job, we can see how everything works together at a higher level:

ma

*Figure 4.4: High-level MapReduce pipeline*

MapReduce inputs typically come from input files loaded onto our processing cluster in HDFS. These files are evenly distributed across all our nodes. Running a MapReduce program involves running mapping tasks on many or all of the nodes in our cluster. Each of these mapping tasks is equivalent: no mappers have particular "identities" associated with them. Therefore, any mapper can process any input file. Each mapper loads the set of files local to that machine and processes them.

When the mapping phase has completed, the intermediate (key, value) pairs must be exchanged between machines to send all values with the same key to a single reducer. The reduce tasks are spread across the same nodes in the cluster as the mappers. **This is the only communication step in MapReduce.** Individual map tasks do not exchange information with one another, nor are they aware of one another's existence. Similarly, different reduce tasks do not communicate with one another. The user never explicitly marshals information from one machine to another; all data transfer is handled by the Hadoop MapReduce platform itself, guided implicitly by the different keys associated with values. This is a fundamental element of Hadoop MapReduce's reliability. If nodes in the cluster fail, tasks must be able to be restarted. If they have been performing *side-effects*, e.g., communicating with the outside world, then the shared state must be restored in a restarted task. By eliminating communication and side-effects, restarts can be handled more gracefully.

### A CLOSER LOOK

The previous figure described the high-level view of Hadoop MapReduce. From this diagram, you can see where the mapper and reducer components of the Word Count application fit in, and how it achieves its objective. We will now examine this system in a bit closer detail.

ma

*Figure 4.5: Detailed Hadoop MapReduce data flow*

Figure 4.5 shows the pipeline with more of its mechanics exposed. While only two nodes are depicted, the same pipeline can be replicated across a very large number of nodes. The next several paragraphs describe each of the stages of a MapReduce program more precisely.

**Input files:** This is where the data for a MapReduce task is initially stored. While this does not need to be the case, the input files typically reside in HDFS. The format of these files is arbitrary; while line-based log files can be used, we could also use a binary format, multi-line input records, or something else entirely. It is typical for these input files to be very large -- tens of gigabytes or more.

**InputFormat:** How these input files are split up and read is defined by the InputFormat. An InputFormat is a class that provides the following functionality:

- Selects the files or other objects that should be used for input
- Defines the *InputSplits* that break a file into tasks
- Provides a factory for *RecordReader* objects that read the file

Several InputFormats are provided with Hadoop. An abstract type is called *FileInputFormat*; all InputFormats that operate on files inherit functionality and properties from this class. When starting a Hadoop job, FileInputFormat is provided with a path containing files to read. The FileInputFormat will read all files in this directory. It then divides these files into one or more InputSplits each. You can choose which InputFormat to apply to your input files for a job by calling the `setInputFormat()` method of the *JobConf* object that defines the job. A table of standard InputFormats is given below.

| InputFormat: | Description: | Key: | Value: |
|---|---|---|---|
| TextInputFormat | Default format; reads lines of text files | The byte offset of the line | The line contents |
| KeyValueInputFormat | Parses lines into key, val pairs | Everything up to the first tab character | The remainder of the line |
| SequenceFileInputFormat | A Hadoop-specific high-performance binary format | user-defined | user-defined |

record. While the TextInputFormat treats the entire line as the value, the KeyValueInputFormat breaks the line itself into the key and value by searching for a tab character. This is particularly useful for reading the output of one MapReduce job as the input to another, as the default OutputFormat (described in more detail below) formats its results in this manner. Finally, the *SequenceFileInputFormat* reads special binary files that are specific to Hadoop. These files include many features designed to allow data to be rapidly read into Hadoop mappers. Sequence files are block-compressed and provide direct serialization and deserialization of several arbitrary data types (not just text). Sequence files can be generated as the output of other MapReduce tasks and are an efficient intermediate representation for data that is passing from one MapReduce job to anther.

**InputSplits:** An InputSplit describes a unit of work that comprises a single *map task* in a MapReduce program. A MapReduce program applied to a data set, collectively referred to as a *Job*, is made up of several (possibly several hundred) tasks. Map tasks may involve reading a whole file; they often involve reading only part of a file. By default, the FileInputFormat and its descendants break a file up into 64 MB chunks (the same size as blocks in HDFS). You can control this value by setting the `mapred.min.split.size` parameter in `hadoop-site.xml`, or by overriding the parameter in the `JobConf` object used to submit a particular MapReduce job. By processing a file in chunks, we allow several map tasks to operate on a single file in parallel. If the file is very large, this can improve performance significantly through parallelism. Even more importantly, since the various blocks that make up the file may be spread across several different nodes in the cluster, it allows tasks to be scheduled on each of these different nodes; the individual blocks are thus all processed locally, instead of needing to be transferred from one node to another. Of course, while log files can be processed in this piece-wise fashion, some file formats are not amenable to chunked processing. By writing a custom InputFormat, you can control how the file is broken up (or is not broken up) into splits. Custom input formats are described in Module 5.

The InputFormat defines the list of tasks that make up the mapping phase; each task corresponds to a single input split. The tasks are then assigned to the nodes in the system based on where the input file chunks are physically resident. An individual node may have several dozen tasks assigned to it. The node will begin working on the tasks, attempting to perform as many in parallel as it can. The on-node parallelism is controlled by the `mapred.tasktracker.map.tasks.maximum` parameter.

**RecordReader:** The InputSplit has defined a slice of work, but does not describe how to access it. The *RecordReader* class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper. The RecordReader instance is defined by the InputFormat. The default InputFormat, *TextInputFormat*, provides a *LineRecordReader*, which treats each line of the input file as a new value. The key associated with each line is its byte offset in the file. The RecordReader is invoke repeatedly on the input until the entire InputSplit has been consumed. Each invocation of the RecordReader leads to another call to the `map()` method of the Mapper.

**Mapper:** The Mapper performs the interesting user-defined work of the first phase of the MapReduce program. Given a key and a value, the `map()` method emits (key, value) pair(s) which are forwarded to the Reducers. A new instance of Mapper is instantiated in a separate Java process for each map task (InputSplit) that makes up part of the total job input. The individual mappers are intentionally not provided with a mechanism to communicate with one another in any way. This allows the reliability of each map task to be governed solely by the reliability of the local machine. The `map()` method receives two parameters in addition to the key and the value:

- The *OutputCollector* object has a method named `collect()` which will forward a (key, value) pair to the reduce phase of the job.
- The *Reporter* object provides information about the current task; its `getInputSplit()` method will return an object describing the current InputSplit. It also allows the map task to provide additional information about its progress to the rest of the system. The `setStatus()` method allows you to emit a status message back to the user. The `incrCounter()` method allows you to increment shared performance counters. You may define as many arbitrary counters as you wish. Each mapper can increment the counters, and the JobTracker will collect the increments made by the different processes and aggregate them for later retrieval when the job ends.

**Partition & Shuffle:** After the first map tasks have completed, the nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers. This process of moving map outputs to the reducers is known as *shuffling*. A different subset of the intermediate key space is assigned to each reduce node; these subsets (known as "partitions") are the inputs to the reduce tasks. Each map task may emit (key, value) pairs to any partition; all values for the same key are always reduced together regardless of which mapper is its origin. Therefore, the map nodes must all agree on where to send the different pieces of the intermediate data. The *Partitioner* class

set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer.

**Reduce:** A Reducer instance is created for each reduce task. This is an instance of user-provided code that performs the second important phase of job-specific work. For each key in the partition assigned to a Reducer, the Reducer's `reduce()` method is called once. This receives a key as well as an iterator over all the values associated with the key. The values associated with a key are returned by the iterator in an undefined order. The Reducer also receives as parameters *OutputCollector* and *Reporter* objects; they are used in the same manner as in the `map()` method.

**OutputFormat:** The (key, value) pairs provided to this OutputCollector are then written to output files. The way they are written is governed by the *OutputFormat*. The OutputFormat functions much like the InputFormat class described earlier. The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS; they all inherit from a common *FileOutputFormat*. Each Reducer writes a separate file in a common output directory. These files will typically be named `part-`*`nnnnn`*, where *nnnnn* is the partition id associated with the reduce task. The output directory is set by the `FileOutputFormat.setOutputPath()` method. You can control which particular OutputFormat is used by calling the `setOutputFormat()` method of the *JobConf* object that defines your MapReduce job. A table of provided OutputFormats is given below.

| OutputFormat: | Description |
|---|---|
| TextOutputFormat | Default; writes lines in "key \t value" form |
| SequenceFileOutputFormat | Writes binary files suitable for reading into subsequent MapReduce jobs |
| NullOutputFormat | Disregards its inputs |

Table 4.2: OutputFormats provided by Hadoop

Hadoop provides some OutputFormat instances to write to files. The basic (default) instance is TextOutputFormat, which writes (key, value) pairs on individual lines of a text file. This can be easily re-read by a later MapReduce task using the *KeyValueInputFormat* class, and is also human-readable. A better intermediate format for use between MapReduce jobs is the *SequenceFileOutputFormat* which rapidly serializes arbitrary data types to the file; the corresponding *SequenceFileInputFormat* will deserialize the file into the same types and presents the data to the next Mapper in the same manner as it was emitted by the previous Reducer. The *NullOutputFormat* generates no output files and disregards any (key, value) pairs passed to it by the OutputCollector. This is useful if you are explicitly writing your own output files in the `reduce()` method, and do not want additional empty output files generated by the Hadoop framework.

**RecordWriter:** Much like how the InputFormat actually reads individual records through the RecordReader implementation, the OutputFormat class is a factory for *RecordWriter* objects; these are used to write the individual records to the files as directed by the OutputFormat.

The **output files** written by the Reducers are then left in HDFS for your use, either by another MapReduce job, a separate program, for for human inspection.

### ADDITIONAL MAPREDUCE FUNCTIONALITY

cor

*Figure 4.6: Combiner step inserted into the MapReduce data flow*

**Combiner:** The pipeline showed earlier omits a processing step which can be used for optimizing bandwidth usage by your MapReduce job. Called the *Combiner,* this pass runs after the Mapper and before the Reducer. Usage of the Combiner is optional. If this pass is suitable for your job, instances of the Combiner class are run on every node that has run map tasks. The Combiner will receive as input all data emitted by the Mapper instances on a given node. The output from the Combiner is then sent to the Reducers, instead of the output from the Mappers. The Combiner is **a "mini-reduce" process which operates only on data generated by one machine.**

Word count is a prime example for where a Combiner is useful. The Word Count program in listings 1--3 emits a `(`*`word`*`, 1)` pair for every instance of every word it sees. So if the same document contains the word "cat" 3 times, the pair `("cat", 1)` is emitted three times; all of these are then sent to the Reducer. By using a Combiner, these can be condensed into a single `("cat", 3)` pair to be sent to the Reducer. Now each node only sends a single value to the reducer for each word -- drastically reducing the total bandwidth required for the shuffle process, and speeding up the job. The best part of all is that we do not need to write any additional

```
conf.setCombinerClass(Reduce.class);
```

The Combiner should be an instance of the *Reducer* interface. If your Reducer itself cannot be used directly as a Combiner because of commutativity or associativity, you might still be able to write a third class to use as a Combiner for your job.

**FAULT TOLERANCE**

One of the primary reasons to use Hadoop to run your jobs is due to its high degree of fault tolerance. Even when running jobs on a large cluster where individual nodes or network components may experience high rates of failure, Hadoop can guide jobs toward a successful completion.

The primary way that Hadoop achieves fault tolerance is through restarting tasks. Individual task nodes (*TaskTrackers*) are in constant communication with the head node of the system, called the *JobTracker*. If a TaskTracker fails to communicate with the JobTracker for a period of time (by default, 1 minute), the JobTracker will assume that the TaskTracker in question has crashed. The JobTracker knows which map and reduce tasks were assigned to each TaskTracker.

If the job is still in the mapping phase, then other TaskTrackers will be asked to re-execute all map tasks previously run by the failed TaskTracker. If the job is in the reducing phase, then other TaskTrackers will re-execute all reduce tasks that were in progress on the failed TaskTracker.

Reduce tasks, once completed, have been written back to HDFS. Thus, if a TaskTracker has already completed two out of three reduce tasks assigned to it, only the third task must be executed elsewhere. Map tasks are slightly more complicated: even if a node has completed ten map tasks, the reducers may not have all copied their inputs from the output of those map tasks. If a node has crashed, then its mapper outputs are inaccessible. So any already-completed map tasks must be re-executed to make their results available to the rest of the reducing machines. All of this is handled automatically by the Hadoop platform.

This fault tolerance underscores the need for program execution to be side-effect free. If Mappers and Reducers had individual identities and communicated with one another or the outside world, then restarting a task would require the other nodes to communicate with the new instances of the map and reduce tasks, and the re-executed tasks would need to reestablish their intermediate state. This process is notoriously complicated and error-prone in the general case. MapReduce simplifies this problem drastically by eliminating task identities or the ability for task partitions to communicate with one another. An individual task sees only its own direct inputs and knows only its own outputs, to make this failure and restart process clean and dependable.

**Speculative execution:** One problem with the Hadoop system is that by dividing the tasks across many nodes, it is possible for a few slow nodes to rate-limit the rest of the program. For example if one node has a slow disk controller, then it may be reading its input at only 10% the speed of all the other nodes. So when 99 map tasks are already complete, the system is still waiting for the final map task to check in, which takes much longer than all the other nodes.

By forcing tasks to run in isolation from one another, individual tasks do not know *where* their inputs come from. Tasks trust the Hadoop platform to just deliver the appropriate input. Therefore, the same input can be processed *multiple times in parallel*, to exploit differences in machine capabilities. As most of the tasks in a job are coming to a close, the Hadoop platform will schedule redundant copies of the remaining tasks across several nodes which do not have other work to perform. This process is known as *speculative execution*. When tasks complete, they announce this fact to the JobTracker. Whichever copy of a task finishes first becomes the definitive copy. If other copies were executing speculatively, Hadoop tells the TaskTrackers to abandon the tasks and discard their outputs. The Reducers then receive their inputs from whichever Mapper completed successfully, first.

Speculative execution is enabled by default. You can disable speculative execution for the mappers and reducers by setting the `mapred.map.tasks.speculative.execution` and `mapred.reduce.tasks.speculative.execution` JobConf options to `false`, respectively.

## Checkpoint

You now know about all of the basic operations of the Hadoop MapReduce platform. Try the following

the word "cat" appears in documents A and B, but not C, then the line:

```
cat    A, B
```

should appear in the output. If the word "baseball" appears in documents B and C, then the line:

```
baseball    B, C
```

should appear in the output as well.

If you get stuck, read the section on troubleshooting below. The working solution is provided at the end of this module.

**Hint:** The default InputFormat will provide the Mapper with (key, value) pairs where the key is the byte offset into the file, and the value is a line of text. To get the filename of the current input, use the following code:

```
FileSplit fileSplit = (FileSplit)reporter.getInputSplit();
String fileName = fileSplit.getPath().getName();
```

## More Tips

**CHAINING JOBS**

Not every problem can be solved with a MapReduce program, but fewer still are those which can be solved with a single MapReduce job. Many problems can be solved with MapReduce, by writing several MapReduce steps which run in series to accomplish a goal:

Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3...

You can easily chain jobs together in this fashion by writing multiple driver methods, one for each job. Call the first driver method, which uses `JobClient.runJob()` to run the job and wait for it to complete. When that job has completed, then call the next driver method, which creates a new `JobConf` object referring to different instances of *Mapper* and *Reducer*, etc. The first job in the chain should write its output to a path which is then used as the input path for the second job. This process can be repeated for as many jobs are necessary to arrive at a complete solution to the problem.

Many problems which at first seem impossible in MapReduce can be accomplished by dividing one job into two or more.

Hadoop provides another mechanism for managing batches of jobs with dependencies between jobs. Rather than submit a `JobConf` to the `JobClient`'s `runJob()` or `submitJob()` methods, `org.apache.hadoop.mapred.jobcontrol.Job` objects can be created to represent each job; A `Job` takes a `JobConf` object as its constructor argument. Jobs can depend on one another through the use of the `addDependingJob()` method. The code:

```
x.addDependingJob(y)
```

says that Job *x* cannot start until *y* has successfully completed. Dependency information cannot be added to a job after it has already been started. Given a set of jobs, these can be passed to an instance of the `JobControl` class. `JobControl` can receive individual jobs via the `addJob()` method, or a collection of jobs via `addJobs()`. The `JobControl` object will spawn a thread in the client to launch the jobs. Individual jobs will be launched when their dependencies have all successfully completed and when the MapReduce system as a whole has resources to execute the jobs. The JobControl interface allows you to query it to retrieve the state of individual jobs, as well as the list of jobs waiting, ready, running, and finished. The job submission process does not begin until the `run()` method of the `JobControl` object is called.

**TROUBLESHOOTING: DEBUGGING MAPREDUCE**

the `logs/` subdirectory of the Hadoop version directory where you run Hadoop from. Log files are named `hadoop-`*`username`*`-service-hostname.log`. The most recent data is in the `.log` file; older logs have their date appended to them. The username in the log filename refers to the username under which Hadoop was started -- this is not necessarily the same username you are using to run programs. The service name refers to which of the several Hadoop programs are writing the log; these can be jobtracker, namenode, datanode, secondarynamenode, or tasktracker. All of these are important for debugging a whole Hadoop installation. But for individual programs, the tasktracker logs will be the most relevant. Any exceptions thrown by your program will be recorded in the tasktracker logs.

The log directory will also have a subdirectory called `userlogs`. Here there is another subdirectory for every task run. Each task records its stdout and stderr to two files in this directory. Note that on a multi-node Hadoop cluster, these logs are not centrally aggregated -- you should check each TaskNode's `logs/userlogs/` directory for their output.

Debugging in the distributed setting is complicated and requires logging into several machines to access log data. If possible, programs should be unit tested by running Hadoop locally. The default configuration deployed by Hadoop runs in "single instance" mode, where the entire MapReduce program is run in the same instance of Java as called `JobClient.runJob()`. Using a debugger like Eclipse, you can then set breakpoints inside the `map()` or `reduce()` methods to discover your bugs.

In Module 5, you will learn how to use additional features of MapReduce to distribute auxiliary code to nodes in the system. This can be used to enable debug scripts which run on machines when tasks fail.

#### LISTING AND KILLING JOBS:

It is possible to submit jobs to a Hadoop cluster which malfunction and send themselves into infinite loops or other problematic states. In this case, you will want to manually kill the job you have started.

The following command, run in the Hadoop installation directory on a Hadoop cluster, will list all the current jobs:

```
$ bin/hadoop job -list
```

This will produce output that looks something like:

```
1 jobs currently running
JobId   State   StartTime       UserName
job_200808111901_0001   1       1218506470390   aaron
```

You can use this job id to kill the job; the command is:

```
$ bin/hadoop job -kill jobid
```

Substitute the "`job_2008...`" from the `-list` command for *jobid*.

## Additional Language Support

Hadoop itself is written in Java; it thus accepts Java code natively for Mappers and Reducers. Hadoop also comes with two adapter layers which allow code written in other languages to be used in MapReduce programs.

#### PIPES

Pipes is a library which allows C++ source code to be used for Mapper and Reducer code. Applications which require high numerical performance may see better throughput if written in C++ and used through Pipes. This library is supported on 32-bit Linux installations.

The include files and static libraries are present in the `c++/Linux-i386-32/` directory under your Hadoop installation. Your application should include `include/hadoop/Pipes.hh` and `TemplateFactory.hh` and

defined in Pipes, are in the `HadoopPipes` namespace.) Unlike the classes of the same names in Hadoop itself, the `map()` and `reduce()` functions take in a single argument which is a reference to an object of type *MapContext* and *ReduceContext* respectively. The most important methods contained in each of these context objects are:

```
const std::string& getInputKey();
const std::string& getInputValue();
void emit(const std::string& key, const std::string& value);
```

The *ReduceContext* class also contains an additional method to advance the value iterator:

```
bool nextValue();
```

**Defining a Pipes Program:** A program to use with Pipes is defined by writing classes extending *Mapper* and *Reducer*. (And optionally, *Partitioner*; see Module 5.) Hadoop must then be informed which classes to use to run the job.

An instance of your C++ program will be started by the Pipes framework in `main()` on each machine. This should do any (hopefully brief) configuration required for your task. It should then define a *Factory* to create Mapper and Reducer instances as necessary, and then run the job by calling the `runTask()` method. The simplest way to define a factory is with the following code:

```
#include"TemplateFactory.hh"
using namespace HadoopPipes;

void main() {
  // classes are indicated to the factory via templates
  // TODO: Substitute your own class names in below.
  TemplateFactory2<MyMapperClass, MyReducerClass> factory();

  // do any configuration you need to do here

  // start the task
  bool result = runTask(factory);
}
```

**Running a Pipes Program:** After a Pipes program has been written and compiled, it can be launched as a job with the following command: (Do this in your Hadoop home directory)

```
$ bin/hadoop pipes -input inputPath -output outputPath -program path/to/pipes/pro
```

This will deploy your Pipes program on all nodes and run the MapReduce job through it. By running `bin/hadoop pipes` with no options, you can see additional usage information which describes how to set additional configuration values as necessary.

The Pipes API contains additional functionality to allow you to read settings from the JobConf, override the Partitioner class, and use RecordReaders in a more direct fashion for higher performance. See the header files in `c++/Linux-i386-32/include/hadoop` for more information.

#### HADOOP STREAMING

Whereas Pipes is an API that provides close coupling between C++ application code and Hadoop, Streaming is a generic API that allows programs written in virtually any language to be used as Hadoop Mapper and Reducer implementations.

The official Hadoop documentation contains a thorough introduction to Streaming, and briefer notes on the wiki. A brief overview is presented here.

Hadoop Streaming allows you to use arbitrary programs for the Mapper and Reducer phases of a MapReduce

programs write their output to stdout in the same format: `key \t value \n`.

The inputs to the reducer are sorted so that while each line contains only a single (key, value) pair, all the values for the same key are adjacent to one another.

Provided it can handle its input in the text format described above, any Linux program or tool can be used as the mapper or reducer in Streaming. You can also write your own scripts in bash, python, perl, or another language of your choice, provided that the necessary interpreter is present on all nodes in your cluster.

**Running a Streaming Job:** To run a job with Hadoop Streaming, use the following command:

```
$ bin/hadoop jar contrib/streaming/hadoop-version-streaming.jar
```

The command as shown, with no arguments, will print some usage information. An example of how to run real commands is given below:

```
$ bin/hadoop jar contrib/streaming-hadoop-0.18.0-streaming.jar -mapper \
    myMapProgram -reducer myReduceProgram -input /some/dfs/path \
    -output /some/other/dfs/path
```

This assumes that *myMapProgram* and *myReduceProgram* are present on all nodes in the system ahead of time. If this is not the case, but they are present on the node launching the job, then they can be "shipped" to the other nodes with the `-file` option:

```
$ bin/hadoop jar contrib/streaming-hadoop-0.18.0-streaming.jar -mapper \
    myMapProgram -reducer myReduceProgram -file \
    myMapProgram -file myReduceProgram -input some/dfs/path \
    -output some/other/dfs/path
```

Any other support files necessary to run your program can be shipped in this manner as well.

## Conclusions

This module described the MapReduce execution platform at the heart of the Hadoop system. By using MapReduce, a high degree of parallelism can be achieved by applications. The MapReduce framework provides a high degree of fault tolerance for applications running on it by limiting the communication which can occur between nodes, and requiring applications to be written in a "dataflow-centric" manner.

## Solution to Inverted Index Code

The following source code implements a solution to the inverted indexer problem posed at the checkpoint. The source code is structurally very similar to the source for Word Count; only a few lines really need to be modified.

```
import java.io.IOException;
import java.util.Iterator;
import java.util.StringTokenizer;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
```

```
    private final static Text word = new Text();
    private final static Text location = new Text();

    public void map(LongWritable key, Text val,
        OutputCollector<Text, Text> output, Reporter reporter)
        throws IOException {

      FileSplit fileSplit = (FileSplit)reporter.getInputSplit();
      String fileName = fileSplit.getPath().getName();
      location.set(fileName);

      String line = val.toString();
      StringTokenizer itr = new StringTokenizer(line.toLowerCase());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        output.collect(word, location);
      }
    }
  }


  public static class LineIndexReducer extends MapReduceBase
      implements Reducer<Text, Text, Text, Text> {

    public void reduce(Text key, Iterator<Text> values,
        OutputCollector<Text, Text> output, Reporter reporter)
        throws IOException {

      boolean first = true;
      StringBuilder toReturn = new StringBuilder();
      while (values.hasNext()){
        if (!first)
          toReturn.append(", ");
        first=false;
        toReturn.append(values.next().toString());
      }

      output.collect(key, new Text(toReturn.toString()));
    }
  }


  /**
   * The actual main() method for our program; this is the
   * "driver" for the MapReduce job.
   */
  public static void main(String[] args) {
    JobClient client = new JobClient();
    JobConf conf = new JobConf(LineIndexer.class);

    conf.setJobName("LineIndexer");

    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(Text.class);

    FileInputFormat.addInputPath(conf, new Path("input"));
    FileOutputFormat.setOutputPath(conf, new Path("output"));

    conf.setMapperClass(LineIndexMapper.class);
    conf.setReducerClass(LineIndexReducer.class);

    client.setConf(conf);

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

Previous module | Table of contents | Next module