1.5 Understanding MapReduce

You're probably aware of data processing models such as pipelines and message queues. These models provide specific capabilities in developing different aspects of data processing applications. The most familiar pipelines are the Unix pipes. Pipelines can help the *reuse* of processing primitives; simple chaining of existing modules creates new ones. Message queues can help the *synchronization* of processing primitives. The programmer writes her data processing task as processing primitives in the form of either a producer or a consumer. The timing of their execution is managed by the system.

Similarly, MapReduce is also a data processing model. Its greatest advantage is the easy scaling of data processing over multiple computing nodes. Under the MapReduce model, the data processing primitives are called *mappers* and *reducers*. Decomposing a data processing application into mappers and reducers is sometimes nontrivial. But, once you write an application in the MapReduce form, scaling the application to run over hundreds, thousands, or even tens of thousands of machines in a cluster is merely a configuration change. This simple scalability is what has attracted many programmers to the MapReduce model.

Many ways to say MapReduce

Even though much has been written about MapReduce, one does not find the name itself written the same everywhere. The original Google paper and the Wikipedia entry use the CamelCase version *MapReduce.* However, Google itself has used *Map Reduce* in some pages on its website (for example, http://research.google.com/roundtable/MR.html). At the official Hadoop documentation site, one can find links pointing to a *Map-Reduce Tutorial*. Clicking on the link brings one to a *Hadoop Map/Reduce Tutorial* (http://hadoop.apache.org/core/docs/current/mapred_tutorial.html) explaining the *Map/Reduce* framework. Writing variations also exist for the different Hadoop components such as *NameNode* (*name node*, *name-node*, and *namenode*), *DataNode*, *JobTracker*, and *TaskTracker*. For the sake of consistency, we'll go with CamelCase for all those terms in this book. (That is, we will use *MapReduce*, *NameNode*, *DataNode*, *JobTracker*, and *TaskTracker*.)

1.5.1 Scaling a simple program manually

Before going through a formal treatment of MapReduce, let's go through an exercise of scaling a simple program to process a large data set. You'll see the challenges of scaling a data processing program and will better appreciate the benefits of using a framework such as MapReduce to handle the tedious chores for you.

Our exercise is to count the number of times each word occurs in a set of documents. In this example, we have a set of documents having only one document with only one sentence:

Do as I say, not as I do.

We derive the word counts shown to the right.

Word	Count
as	2
do	2
i	2
not	1
say	1

We'll call this particular exercise *word counting*. When the set of documents is small, a straightforward program will do the job. Let's write one here in pseudo-code:

```
define wordCount as Multiset;
for each document in documentSet {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
display(wordCount);
```

The program loops through all the documents. For each document, the words are extracted one by one using a tokenization process. For each word, its corresponding entry in a multiset called `wordCount` is incremented by one. At the end, a `display()` function prints out all the entries in `wordCount`.

> **NOTE** A multiset is a set where each element also has a count. The word count we're trying to generate is a canonical example of a multiset. In practice, it's usually implemented as a hash table.

This program works fine until the set of documents you want to process becomes large. For example, you want to build a spam filter to know the words frequently used in the millions of spam emails you've received. Looping through all the documents using a single computer will be extremely time consuming. You speed it up by rewriting the program so that it distributes the work over several machines. Each machine will process a distinct fraction of the documents. When all the machines have completed this, a second phase of processing will combine the result of all the machines. The pseudo-code for the first phase, to be distributed over many machines, is

```
define wordCount as Multiset;
for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}
sendToSecondPhase(wordCount);
```

And the pseudo-code for the second phase is

```
define totalWordCount as Multiset;
for each wordCount received from firstPhase {
    multisetAdd (totalWordCount, wordCount);
}
```

That wasn't too hard, right? But a few details may prevent it from working as expected. First of all, we ignore the performance requirement of reading in the documents. If the documents are all stored in one central storage server, then the bottleneck is in the bandwidth of that server. Having more machines for processing only helps up to a certain point—until the storage server can't keep up. You'll also need to split up the documents among the set of processing machines such that each machine will process only those documents that are stored in it. This will remove the bottleneck of a central storage server. This reiterates the point made earlier about storage and processing having to be tightly coupled in data-intensive distributed applications.

Another flaw with the program is that `wordCount` (and `totalWordCount`) are stored in memory. When processing large document sets, the number of unique words can exceed the RAM storage of a machine. The English language has about one million words, a size that fits comfortably into an iPod, but our word counting program will deal with many unique words not found in any standard English dictionary. For example, we must deal with unique names such as *Hadoop*. We have to count misspellings even if they are not real words (for example, *exampel*), and we count all different forms of a word separately (for example, *eat*, *ate*, *eaten*, and *eating*). Even if the number of unique words in the document set is manageable in memory, a slight change in the problem definition can explode the space complexity. For example, instead of words

in documents, we may want to count IP addresses in a log file, or the frequency of bigrams. In the case of the latter, we'll work with a multiset with billions of entries, which exceeds the RAM storage of most commodity computers.

> **NOTE** A bigram is a pair of consecutive words. The sentence "Do as I say, not as I do" can be broken into the following bigrams: *Do as*, *as I*, *I say*, *say not*, *not as*, *as I*, *I do*. Analogously, trigrams are groups of three consecutive words. Both bigrams and trigrams are important in natural language processing.

`wordCount` may not fit in memory; we'll have to rewrite our program to store this hash table on disk. This means we'll implement a disk-based hash table, which involves a substantial amount of coding.

Furthermore, remember that phase two has only one machine, which will process `wordCount` sent from *all* the machines in phase one. Processing one `wordCount` is itself quite unwieldy. After we have added enough machines to phase one processing, the single machine in phase two will become the bottleneck. The obvious question is, can we rewrite phase two in a distributed fashion so that it can scale by adding more machines?

The answer is, yes. To make phase two work in a distributed fashion, you must somehow divide its work among multiple machines such that they can run independently. You need to *partition* `wordCount` after phase one such that each machine in phase two only has to handle one partition. In one example, let's say we have 26 machines for phase two. We assign each machine to only handle `wordCount` for words beginning with a particular letter in the alphabet. For example, machine A in phase two will only handle word counting for words beginning with the letter *a*. To enable this partitioning in phase two, we need a slight modification in phase one. Instead of a single disk-based hash table for `wordCount`, we will need 26 of them: `wordCount-a`, `wordCount-b`, and so on. Each one counts words starting with a particular letter. After phase one, `wordCount-a` from each of the phase one machines will be sent to machine A of phase two, all the `wordCount-b`'s will be sent to machine B, and so on. Each machine in phase one will *shuffle* its results among the machines in phase two.

Looking back, this word counting program is getting complicated. To make it work across a cluster of distributed machines, we find that we need to add a number of functionalities:

- Store files over many processing machines (of phase one).
- Write a disk-based hash table permitting processing without being limited by RAM capacity.
- Partition the intermediate data (that is, `wordCount`) from phase one.
- Shuffle the partitions to the appropriate machines in phase two.

This is a lot of work for something as simple as word counting, and we haven't even touched upon issues like fault tolerance. (What if a machine fails in the middle of its task?) This is the reason why you would want a framework like Hadoop. When you

write your application in the MapReduce model, Hadoop will take care of all that scalability "plumbing" for you.

1.5.2 Scaling the same program in MapReduce

MapReduce programs are executed in two main phases, called *mapping* and *reducing*. Each phase is defined by a data processing function, and these functions are called *mapper* and *reducer*, respectively. In the mapping phase, MapReduce takes the input data and feeds each data element to the mapper. In the reducing phase, the reducer processes all the outputs from the mapper and arrives at a final result.

In simple terms, the mapper is meant to *filter and transform* the input into something that the reducer can *aggregate* over. You may see a striking similarity here with the two phases we had to develop in scaling up word counting. The similarity is not accidental. The MapReduce framework was designed after a lot of experience in writing scalable, distributed programs. This two-phase design pattern was seen in scaling many programs, and became the basis of the framework.

In scaling our distributed word counting program in the last section, we also had to write the partitioning and shuffling functions. Partitioning and shuffling are common design patterns that go along with mapping and reducing. Unlike mapping and reducing, though, partitioning and shuffling are generic functionalities that are not too dependent on the particular data processing application. The MapReduce framework provides a default implementation that works in most situations.

In order for mapping, reducing, partitioning, and shuffling (and a few others we haven't mentioned) to seamlessly work together, we need to agree on a common structure for the data being processed. It should be flexible and powerful enough to handle most of the targeted data processing applications. MapReduce uses *lists* and *(key/value) pairs* as its main data primitives. The keys and values are often integers or strings but can also be dummy values to be ignored or complex object types. The map and reduce functions must obey the following constraint on the types of keys and values.

	Input	Output
map	<k1, v1>	list(<k2, v2>)
reduce	<k2, list(v2)>	list(<k3, v3>)

In the MapReduce framework you write applications by specifying the mapper and reducer. Let's look at the complete data flow:

1. The input to your application must be structured as a list of (key/value) pairs, `list(<k1, v1>)`. This input format may seem open-ended but is often quite simple in practice. The input format for processing multiple files is usually `list(<String filename, String file_content>)`. The input format for processing one large file, such as a log file, is `list(<Integer line_number, String log_event>)`.

2. The list of (key/value) pairs is broken up and each individual (key/value) pair, `<k1, v1>`, is processed by calling the map function of the mapper. In practice, the key `k1` is often ignored by the mapper. The mapper transforms each `<k1, v1>` pair into a list of `<k2, v2>` pairs. The details of this transformation largely determine what the MapReduce program does. Note that the (key/value) pairs are processed in arbitrary order. The transformation must be self-contained in that its output is dependent only on one single (key/value) pair.

   For word counting, our mapper takes `<String filename, String file_content>` and promptly ignores `filename`. It can output a list of `<String word, Integer count>` but can be even simpler. As we know the counts will be aggregated in a later stage, we can output a list of `<String word, Integer 1>` with repeated entries and let the complete aggregation be done later. That is, in the output list we can have the (key/value) pair `<"foo", 3>` once or we can have the pair `<"foo", 1>` three times. As we'll see, the latter approach is much easier to program. The former approach may have some performance benefits, but let's leave such optimization alone until we have fully grasped the MapReduce framework.
3. The output of all the mappers are (conceptually) aggregated into one giant list of `<k2, v2>` pairs. All pairs sharing the same `k2` are grouped together into a new (key/value) pair, `<k2, list(v2)>`. The framework asks the reducer to process each one of these aggregated (key/value) pairs individually. Following our word counting example, the map output for one document may be a list with pair `<"foo", 1>` three times, and the map output for another document may be a list with pair `<"foo", 1>` twice. The aggregated pair the reducer will see is `<"foo", list(1,1,1,1,1)>`. In word counting, the output of our reducer is `<"foo", 5>`, which is the total number of times "foo" has occurred in our document set. Each reducer works on a different word. The MapReduce framework automatically collects all the <k3, v3> pairs and writes them to file(s). Note that for the word counting example, the data types `k2` and `k3` are the same and `v2` and `v3` are also the same. This will not always be the case for other data processing applications.

Let's rewrite the word counting program in MapReduce to see how all this fits together Listing 1.1 shows the pseudo-code.

Listing 1.1 Pseudo-code for map and reduce functions for word counting

```
map(String filename, String document) {
    List<String> T = tokenize(document);
    for each token in T {
        emit ((String)token, (Integer) 1);
    }
}
reduce(String token, List<Integer> values) {
    Integer sum = 0;
```

```
    for each value in values {
        sum = sum + value;
    }
    emit ((String)token, (Integer) sum);
}
```

We've said before that the output of both map and reduce function are lists. As you can see from the pseudo-code, in practice we use a special function in the framework called `emit()` to generate the elements in the list one at a time. This `emit()` function further relieves the programmer from managing a large list.

The code looks similar to what we have in section 1.5.1, except this time it will actually work at scale. Hadoop makes building scalable distributed programs easy, doesn't it? Now let's turn this pseudo-code into a Hadoop program.