Chapter 8

Cloud Computing

In **cloud computing**, the idea is that a large corporation that has many computers could sell time on them, for example to make profitable use of excess capacity. The typical customer would have occasional need for large-scale computing—and often large-scale data storage. The customer would submit a program to the cloud computing vendor, who would run it in parallel on the vendor's many machines (unseen, thus forming the "cloud"), then return the output to the customer.

Google, Yahoo! and Amazon, among others, have recently gotten into the cloud computing business. Moreover, universities, businesses, research labs and so on are setting up their own small clouds, typically on clusters (a bunch of computers on the same local network, possibly with central controlling software for job management).

The paradigm that has become standard in the cloud today is MapReduce, developed by Google. In rough form, the approach is as follows. Various nodes server as *mappers*, and others serve as *reducers*.[1]

The terms *map* and **reduce** are in the functional programming sense. In the case of **reduce**, the idea is similar to reduction oeprations we've seen earlier in this book, such as the **reduction** clause in OpenMP and **MPI_Reduce()** for MPI. So, reducers in Hadoop perform operations such as summation, finding minima or maxima, and so on.

8.1 Hadoop

In this chapter we give a very brief introduction to Hadoop, today's open-source application of choice of MapReduce.

[1] Of course, some or all of these might be threads on the same machine.

8.2 Platforms

In terms of platforms, Hadoop is basically a Linux product. Quoting from the Hadoop Quick Start, `http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html#Supported+Platforms`:

> Supported Platforms:
>
> - GNU/Linux is supported as a development and production platform. Hadoop has been demonstrated on GNU/Linux clusters with 2000 nodes.
> - Win32 is supported as a *development platform*. Distributed operation has not been well tested on Win32, so it is not supported as a *production platform*.

8.3 Overview of Operations

Here is a sumary of how a Hadoop application runs:

- Divide the input data into chunks of records. (In many cases, this is already the case, as a very large data file might be distributed across many machines.)
- Send the chunks to the mappers.
- Each mapper does some transformation (a "map," in functional programming terms) to each record in its chunks, and sends the output records to Hadoop.
- Hadoop collects the transformed records, splits them into chunks, sorts them, and then sends the chunks to the reducers.
- Each reducer does some summary operation (functional programming "reduce"), producing output records.
- Hadoop collects all those output records and does the same summary operation to them, producing the final output.

8.4 Role of Keys

The sorting operation, called the **shuffle** phase, is based on a **key** defined by the programmer. The key defines groups. If for instance we wish to find the total number of men and women in a certain debate, the key would be gender. The reducer would do addition, in this case adding 1s, one 1 for each person, but keeping a separate sum for each gender.

During the shuffle stage, Hadoop sends all records for a given key, e.g. all men, to one reducer. In other words, records for a given key will never be split across multiple reducers. (Keep in mind, though, that typically a reducer will have the records for many keys.)

8.5 Hadoop Streaming

Actually Hadoop is really written for Java or C++ applications. However, Hadoop can work with programs in any language under Hadoop's Streaming option, by reading from STDIN and writing to STDOUT, in text, line-oriented form in both cases. In other words, any executable program, be it Java, C/C++, Python, R, shell scripts or whatever, can run in Hadoop in streaming mode.

Everything is text-file based. Mappers input lines of text, and output lines of text. Reducers input lines of text, and output lines of text. The final output is lines of text.

Streaming mode may be less efficient, but it is simple to develop programs in it, and efficient enough in many applications. Here we present streaming mode.

So, STDIN and STDOUT are key in this mode, and as mentioned earlier, input and output are done in terms of lines of text. One additional requirement, though, is that the line format for both mappers and reducers must be

```
key \t value
```

where \t is the TAB character.

The usage of text format does cause some slowdown in numeric programs, for the conversion of strings to numbers and vice versa, but again, Hadoop is not designed for maximum efficiency.

8.6 Example: Word Count

The typical introductory example is word count in a group of text files. One wishes to determine what words are in the files, and how many times each word appears. Let's simplify that a bit, so that we simply want a count of the number of words in the files, not an individual count for each word.

The initial input is the lines of the files (combined internally by Hadoop into one superfile). The mapper program breaks a line into words, and emits (key,value) pairs in the form of (0,1). Our key here, 0, is arbitrary and meaningless, but we need to have one.

In the reducer stage, all those (key,value) pairs get sorted by the Hadoop internals (which has no effect in this case), and then fed into the reducers. The latter add up all their input values, i.e. all

the 1s, yielding a sum for each reducer. Hadoop combines thus sums, applying the same reduce operation to them, yielding a grand total number of words in all the files.

Here's the pseudocode:

mapper:

```
1  for each line in STDIN
2     break line into words, placed in wordarray
3     for each word in wordarray
4        # we have found 1 word
5        print '0', '1' to STDOUT
```

reducer:

```
1  count = 0
2  for each line in STDIN
3     split line into (key,value) # i.e. (0,1) here
4     count += value   # i.e. add 1 to count
5  print key, count  # i.e. 0 and count
```

So, each reducer will print out a single line, consisting of 0 and the count of the words it was fed. Hadoop will collect these lines, and run the same reducer on them, yielding 0 and a final count.

In terms of the key 0, the final output tells us how many words there were of type 0. Since we arbitrarily considered all words to be of type 0, the final output is simply an overall word count.

8.7 Example: Maximum Air Temperature by Year

A common Hadoop example on the Web involves data with the format for each line

```
year month day high_temperature air_quality
```

It is desired to find the maximum air temperature for each year.

The code would be very similar to the outline in the word count example above, but now we have a real key—the year. So, in the end, the final output will be a listing of maximum temperature by year.

Our mapper here will simply be a text processor, extracting year and temperature from each line of text. (This is a common paradigm for Hadoop applications.) The reducer will then do the maximum-finding work.

Here is the pseudocode:

mapper:

```
1  for each line in STDIN
2     extract year and temperature
3     print year, temperature to STDOUT
```

We have to be a bit more careful in the case of the reducer. Remember, though no year will be split across reducers, each reducer will likely receive the data for more than one year. It needs to find and output the maximum temperature for each of those years.[2]

Since Hadoop internals sort the output of the mappers by key, our reducer code can expect a bunch of records for one year, then a bunch for another year and so on. So, as the reducer goes through its input line by line, it needs to detect when one bunch ends and the next begins. When such an event occurs, it outputs the max temp for the bunch that just ended.

Here is the pseudocode:

reducer:

```
1  currentyear = NULL
2  currentmax = "-infinity"
3  for each line in STDIN
4     split line into year, temperature
5     if year == currentyear:  # still in the current bunch
6        currentmax = max(currentmax,temperature)
7     else:  # encountered a new bunch
8        # print summary for previous bunch
9        if currentyear not NULL:
10          print currentyear, currentmax
11       # start our bookkeeping for the new bunch
12       currentyear = year
13       currentmax = temperature
14 print currentyear, currentmax
```

8.8 Role of Disk Files

Hadoop has its own file system, HDFS, which is built on top of the native OS' file system of the machines. It is replicated for the sake of reliability, with each HDFS block existing in at least 3 copies, i.e. on at least 3 separate disks.

Very large files are possible, in some cases spanning more than one disk/machine. Indeed, this is the typical goal of Hadoop—to easily parallelize operations on a very large database.

Disk files play a major role in Hadoop programs:

- Input is from a file in the HDFS system.

[2]The results from the various reducers will then in turn be reduced, yielding the max temps for all years.

- The output of the mappers goes to temporary files in the native OS' file system.
- Final output is to a file in the HDFS system. As noted earlier, that file may be distributed across several disks/machines.

Note that by having the input and output files in HDFS, we minimize communications costs in shipping the data. The slogan used is "Moving computation is cheaper than moving data."

8.9 The Hadoop Shell

The HDFS can be accessed via a set of Unix-like commands. For example,

```
1 hadoop fs -mkdir somedir
```

creates a file **somedir** in my HDFS (invisible to me). Then

```
1 hadoop fs -put gy* somedir
```

copies all my files whose names begin with "gy" to **somedir**, and

```
1 hadoop fs -ls somedir
```

lists the file names in the directory **somedir**.

See `http://hadoop.apache.org/common/` for a list of available commands.

8.10 Running Hadoop

You run the above word count example something like this, say on the UCD CSIF machines.

Say my input data is in the directory **indata** on my HDFS, and I want to write the output to a new directory **outdata**. Say I've placed the mapper and reducer programs in my home directory (non-HDFS). I could then run

```
$ hadoop jar \
    /usr/local/hadoop-0.20.2/contrib/streaming/hadoop-0.20.2-streaming.jar \
     -input indata -output outdata \
     -mapper mapper.py -reducer reducer.py \
     -file /home/matloff/mapper.py \
     -file /home/matloff/reducer.py
```

This tells Hadoop to run a Java **.jar** file, which in our case here contains the code to run streaming-mode Hadoop, with the specified input and output data locations, and with the specified mapper

and reducer functions. The **-file** flag indicates the locations of those functions (not needed if they are in my shell search path).

I could then run

```
1 hadoop fs -ls outdata
```

to see what files were produced, say **part_00000**, and then type

```
1 hadoop fs -cat outdata/part_00000
```

to view the results.

Note that the **.py** files must be executable, both in terms of file permissions and in terms of invoking the Python interpreter, the latter done by including

```
#!/usr/bin/env python
```

as the first line in the two Python files.

8.11 Example: Transforming an Adjacency Graph

Yet another rendition of the app in Section 4.13, but this time with a bit of problem, which will illustrate a limitation of Hadoop.

To review:

Say we have a graph with adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{8.1}$$

with row and column numbering starting at 0, not 1. We'd like to transform this to a two-column

matrix that displays the links, in this case

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{8.2}$$

Suppose further that we require this listing to be in lexicographical order, sorted on source vertex and then on destination vertex.

At first, this seems right up Hadoop's alley. After all, Hadoop does sorting for us within groups automatically, and we could set up one group per row of the matrix, in other words make the row number the key.

We will actually do this below, but there is a fundamental problem: Hadoop's simple elegance hinges on there being an independence between the lines in the input file. We should be able to process them one line at a time, independently of other lines.

The problem with this is that we will have many mappers, each reading only some rows of the adjacency matrix. Then for any given row, the mapper handling that row doesn't know what row number this row had in the original matrix. So we have no key!

The solution is to add a column to the matrix, containing the original row numbers. The matrix above, for instance, would become

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 2 & 0 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{8.3}$$

Adding this column may be difficult to do, if the matrix is very large and already distributed over many machines. Assuming we do this, though, here is the mapper code (real code this time, not pseudocode):

```
1 #!/usr/bin/env python
2
3 # map/reduce pair inputs a graph adjacency matrix and outputs a list of
4 # links; if say row 3, column 8 of the input is 1, then there will be a
5 # row (3,8) in the final output
6
```

```
7  import sys
8
9  for line in sys.stdin:
10     tks = line.split() # get tokens
11     srcnode = tks[0]
12     links = tks[1:]
13     for dstnode in range(len(links)):
14        if links[dstnode] == '1':
15            toprint = '%s\t%s' % (srcnode, dstnode)
16            print toprint
```

And the reducer code:

```
1  #!/usr/bin/env python
2
3  import sys
4
5  for line in sys.stdin:
6      line = line.strip()
7      srcnode, dstnode = line.split('\t')
8      print '%s\t%s' % (srcnode, dstnode)
```

Recall that in the word count and yearly temperature examples above, the reducer did the main work, with the mappers playing only a supporting role. In this case here, it's the opposite, with the reducers doing little more than printing what they're given. *However*, keep in mind that Hadoop itself did a lot of the work, with its shuffle phase, which produced the sorting that we required in the output.

8.12 Debugging Hadoop Streaming Programs

One advantage of the streaming approach is that mapper and reducer programs can be debugged via normal tools, since those programs can be run on their own, without Hadoop, simply by using the Unix/Linux pipe capability.

This all depends on the fact that Hadoop essentially does the following shell computation:

```
cat inputfile | mapperprog | sort | reducerprog
```

The mapper alone works like

```
cat inputfile | mapperprog
```

You thus can use whatever debugging tool you favor, to debug the mapper and reducer code separately.

Note, though, that the above pipe is *not quite* the same as Hadoop. the pipe doesn't break up the data, and there may be subtle problems arising as a result. But overall, the above approach provides a quick and easy first attempt at debugging.