6.033
COMPUTER SYSTEMS ENGINEERING
HANDS ON 4: MAPREDUCE

JOHN WANG

1. Studying mapreduce.py

(1) The first two parameters of WordCount are the number of map tasks and reduce tasks that the WordCount job will span, respectively. The number of map tasks splits up the inputs into that many temporary files, and the number of reduce tasks splits up the work into that many different processes.

(2) The run() method is called from WordCount's superclass, which is the MapReduce class. The run() method first calls the doMap() method for all of the map tasks, then proceeds to call the doReduce() method for all the reduce tasks. The doMap() method works by taking in the split input and calling the Map() function defined in WordCount on that input, and dumping the results into a file so that the doReduce() method can read them and reduce everything. After all the doMap() jobs are finished, the doReduce() methods begin. These methods take in the files that were created during the doMap() phase, and call Reduce() on the files as defined in the WordCount function.

(3) The keyvalue is byte offset from the beginning of the bible file. It serves as a key to the portion of the bible file which the Map() job will be considering. The value is the portion of the bible file which the Map() job is running over.

(4) The key represents the word which is found inside of the bible file, and the keyvalues represent a list of tuples. Each tuple in the list inside keyvalues contains the word as the first item, and the number of occurrences of the word as the second item.

2. Modifying mapreduce.py

(5) There are 4 calls to doMap() and 2 calls to doReduce(). This occurs because these values were the values set in our initialization of the WordCount object to be the number of map and reduce jobs, respectively, that the WordCount object would spawn. Recall that the first two parameters in the WordCount object initialization corresponded to the number of map and reduce jobs to start.

(6) All of the doMap() jobs should run in parallel. In addition, the doReduce() jobs should also run in parallel. This is the case because the code that calls these jobs is the following:

```
regions = pool.map(self.doMap, range(0, self.maptask))
partitions = pool.map(self.doReduce, range(0, self.reducetask))
```

These two commands spawn multiple processes to carry out the doMap commands, and also the doReduce commands. They spawn a number of processes equal to the number of map and reduce jobs that were set at the beginning of the WordCount object instantiation.

(7) A single doMap() processes about 1208690 bytes of input, but this number changes by a couple bytes because the inputs are split at the first whitespace character that is greater than the chunk size of 1208690.

(8) A single doReduce() processes about 2250 keys, but this number is not exact and depends on the number of title words in the associated map jobs for each doReduce() job.

(9) The table below shows a number of trials for different parameters. I observe speedup for a medium number of map and reduce jobs. For large numbers of map and reduce jobs (for example 13 map jobs), I saw significant performance dropoffs. This may have resulted from the fact that my machine only has 4 cores and was therefore unable to take advantage of the concurrency in the program.

3. Reverse Word Index

(10) It took me two hours and 30 minutes to complete the assignment.

		# Reduce Jobs				
		1	4	7	10	13
# Map Jobs	1	18.6	20.2	16.8	16.4	15.8
	4	15.7	16.1	17.1	16.3	16.8
	7	17.5	18.3	18.1	18.4	19.6
	10	20.0	18.9	19.8	19.1	18.4
	13	21.9	48.0	41.3	31.1	21.8

FIGURE 1. Seconds required to count words for different number of map and reduce jobs. Note that 3 trials were run for each set of map and reduce jobs.