SE227
COMPUTER SYSTEMS ENGINEERING
HANDS ON 3: MAPREDUCE

QINGQING CHENG

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: the number of temporary files that the inputs is split up into; the number of reduce tasks: the number of different processes that the work is split up into.

(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	14.7742311954	15.1326069832	14.291934967	14.2608690262	15.0647192001
	4	14.3060090542	14.5823261738	14.5286550522	15.6719620228	14.8089988232
	7	14.6576430798	15.3284409046	16.0787160397	16.0909619331	16.4534978867
	10	16.3597910404	16.4728901386	16.2503421307	15.9733150005	15.5352330208
	13	15.5517809391	16.5351829529	16.3348960876	17.2354898453	16.656111002

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.