6.033 Mapreduce

Kevin Cho
R5 Peter Szolovits 1PM

Due February 16th 2016

I Studying mapreduce.py

1. The first parameter that goes into WordCount init() is maptask. Maptask is the number of map tasks that needs to be processed. In other words, the number of given in map tasks is the how many tasks the inputs are split into for the mapping process. The second paramter is the reducetask. This is the same as maptasks, but it is the number of reduce tasks that we give the function. In other words, the number given to reducetask, is the number of reducing tasks our WordCount will do.

2. The run function is a method from the superclass of MapReduce. What the run function does first is create a pool with the number of maptasks and reducetasks. Now, it calls doMap on the range of maptasks and a doReduce on range of reducetasks. The way doMap works is that it takes in an input that is alreadsy split and calls the Map function from WordCount on that input. Then, doMap will put everything in a file so that doReduce can read and reduce whatever is on that file. The doMap goes on for all the jobs, then the doReduce part starts. The doReduce is similar to the doMap in that it calls Reduce function in WordCount instead of the Map. This is how the run function connects the Map and Reduce functions.

3. The keyvalue can be seen as the byte offset from the beginning of the file. It is essentially the key of the file portion that the Map function is running on. The value field is the portion of the file where the map function is actually running on.

4. The key is the word that is found in the file that we want to count. The keyvalues is a list that contains how many times the word occured in the file. All the numbers in keyvalues essentially are one until combined. Each item of the keyvalues list is a tuple where the first value is the key or the word, and the second value is an integer, which is the number of occurences.

II Modifying mapreduce.py

5. The current set up has us run doMap 4 times and doReduce 2 times. This is because on the initialization of WordCount, we set maptask to be 4 and reducetask to be 2. This means that the number of map tasks is 4 thus runs doMap 4 times, and the number of reduce tasks is 2 thus runs doReduce 2 times.

6. For this question, we can take a look at the actual run() function code. When we set up the pool, we set it up so that we have the max of maptasks and reducetasks as the Pool number. Thus, with this pool we map using the doMap function, which this process would run in parallel. We also have the pool map function used on doReduce for all its iterations, thus these run in parallel. Thus, all these processes run in parallel because the pool.map() spawns enough cores or processes to processes all these iterations in parallel.

7. Each doMap process processes about 1208595 bytes or letters. However, we have to remember that these are all the bytes including the whitespaces. Thus, the actual amount of letters will be much smaller than that, but the bytes are the same.

8. Each doReduce process processes about 2250 keys. The exact numbers were 2247 and 2221 keys for both processes, respectively. However, this number does depend on the number of ittel words in the associated map jobs.

9. After some testing with different numbers for the reduce jobs and map jobs, I found that having about a medium or reasonable amount of jobs for both provided the best overall speed. I think the reason behind this is that once the jobs get too big, it runs much slower because we are limited on the number of cores that the process runs on. Thus, by having large number of jobs, we would start to notice the "speedup" actually be a slower process.

10. Here is the code for my ReverseIndex:

```
class ReverseIndex(MapReduce):
    def __init__(self, maptask, reducetask, path):
        MapReduce.__init__(self,  maptask, reducetask, path)

    # Produce a (key, value) pair for each title word in value
    def Map(self, keyvalue, value):
        results = []
        i = 0
        n = len(value)
        keyvalue = keyvalue[:-1]
        while i < n:
            # skip non-ascii letters in C/C++ style a la MapReduce paper:
            while i < n and value[i] not in string.ascii_letters:
                i += 1
            start = i
            while i < n and value[i] in string.ascii_letters:
                i += 1
            w = value[start:i]
            keyvalue1 = int(keyvalue) + start
            if start < i:
                results.append ((w.lower(), keyvalue1))
        return results

    # Reduce [(key,value), ...])
    def Reduce(self, key, keyvalues):
        index = []
        for pair in keyvalues:
            index.append(pair[1])
        return (key, index)
```

This assignment took me about 3 hours to finish.