Exercises for Hands-on 4: Map-Reduce

Make sure to fill in questions in the space provided underneath the question text.

Studying mapreduce.py

Question 1: The parameters to `WordCount's __init__` method are `maptask` and `reducetask`.

a) What does `maptask` control?

b) What does `reducetask` control?

Question 2: Briefly explain how calling run triggers calls to both the map and reduce methods of the WordCount instance.

---

1) Maptask controls the number of map tasks created, and also serves as an index. This integer also is used to divide the input into 'chunks.'
Reducetask also control the number of tasks and serves as an index.

2) When you call WordCount.run(), it invokes the .run() method of the parent class, MapReduce. On initialization of the parent class, self.Split() is called and the input file is split into `maptask` different 'chunks'. The run() method begins a pool of processes, then invokes doMap() and doReduce() on the processes with the pickled splits from the spit() method, and passes them an index, i, from the maptask/reducetask variables. The 'do' methods then call self.map() and self.reduce() methods of the child class respectively, and then they the resultant data into byte streams which are "pickled" into files on the local machine's filesystem, not inside the object.

Question 3: What do the parameters `keyvalue` and `value` of the `map` method in WordCount represent?
Question 4: What do the parameters `key` and `keyvalues` of the `reduce` method in WordCount represent?

---

3) **`value`** represents the text to be viewed as a string. Specifically, It is a string representation of the bible or chunk of it.
`keyvalue` represents the byte offset into the original file. Concretely, you can see in MapReduce.split() that `keyvalue` is stored as the first line of the temp pickle file, and is the number of chars written from the buffer to the '#split ….' files in all. Though, `keyvalue` is never called in WordCount.map(), it is needed in reverseindex.py in order to differentiate the offset into the chunk, vs the offset into the original input file.

4) **`key`** represents the word to be counted. This is the mutual word in `keyvalues`.

`keyvalues` is composed in the pre-processing method doReduce. doReduce loads the 'chunk' of counted words from the temp pickle file, where the words are represented as tuples of ("word",1). `keyvalues` represents a list of tuples which have the same "word" (equal to the `key` parameter) in the pickle. The reduce() method counts the tuples.

Modifying mapreduce.py

Modify the program so that you can answer the following questions:

Question 5: How many invocations are there to `doMap` and how many to `doReduce`? Why?
Question 6: Which invocations run in parallel? (Assuming there are enough cores.)

---

5) doMap and doReduce are both invoked 2 times. MapReduce.split() splits the input into 2 chunks (presumable so we can run it on our dual-core processors). In run(), you can see that we Pool 2 processes for doMap(), then Pool 2 processes for doReduce().

6) Assuming enough cores, the doMap()'s will run in parallel and the doReduce's then run in parallel afterward. Empirically, we can add print statements at the beginning of the methods in question, and we can see the console log that doMap() is invoked twice , then doReduce() is invoked twice. Intuitively, split() has already divided the input into 'chunks', so running the mapping function on the 2 chunks concurrently would have no side effects. Conversely, running map in parallel with reduce makes no sense, the intermediate may not be ready. Additionally, if we add print statements inside of the methods, we can observe that the print statements at the beginning of the method are both printed before any statement at the end of the method. Thus, they must be running in parallel.

Question 7: How much input (in number of bytes) does a single `doMap` process?
Question 8: How much input (in number of keys) does a single `doReduce` process?

---

7) Implicitly, we can read the first line of the split pickle file, for the byte-offset recorded from the number of chars written. When we do this, we get: 0 and 2417380. Meaning the first split goes from 0-2417380, and the 2nd from 2417380-> 2*2417380.
Explicitly, we can also add the line `print os.stat("#split-%s-%s" % (self.path, i)).st_size` to the beginning of doMap(), which yeilds: **2417382 and 2417385** as the number of bytes for the input file at each invocation.

8) If we add a counter under the line `for item in itemlist:`, we will count all the times we view an item from file, where each is a key. This yields:
32807 and 55288 items assigned to keys for the 2 processes of doReduce(). We can also view the number of output keys, which will be significantly lower as many input keys map to the same output keys. When looking at the len(keys.keys()), we get:
2250 and 2222.

Question 9: For which parameters of `maptask` and `reducetask` do you see speedup? Why do you observe no speedup for some parameters? (You might see no speedup at all on a busy machine or a machine with a single core.)
Question 10: What is the bottleneck of this process? If you wanted to decrease the amount of time it took to run, what would you do?

---

9) If we use the command `$ time python mapreduce.py kjv12.txt `

maptask	**reducetask**	**Time**
1	1	0m3.862s
2	1	0m3.370s
2	2	0m2.869s
3	3	0m2.867s
4	4	0m2.868s
2	4	0m2.879s
4	2	0m2.902s
8	8	0m2.995

We only observe a speedup when going from 1 process to 2 processes. Given that my local machine has 2 physical cores (4 virtual), we see that the parameters of (2,2) and (4,4) work the best. With using a parameter of 1, we only get 1 worker, and therefore get none of the benefits of parallelization for mapreduce. Since we don't have any more cores to use, any additional workers would be sharing time on one of the cores, and would have to be switching contexts, timesharing, and not actually being completely parallel.

10) The bottleneck here is the fact my i5 only has 2 physical cores, and therefore 2 actual workers that could perform tasks without time sharing. If we wanted to improve performance, we would need more cores/machines/CPUs to act as workers.

To further test this hypothesis (and because, why not), I spun up a 64-core Haswell instance on Google Compute Engine, and re-tried the above task on the 64 vCPU machine. Here, we see continued speed up past the 2 cores, but eventually levels off at between 16-32 cores, most likely due to reads/writes not being able to be done any faster.

#(maptask)	**#(reducetask)**	**time**
1	1	0m3.053s
2	2	0m2.199s
4	4	0m1.797s
8	8	0m1.590s
16	16	0m1.384s
32	32	0m1.357s
64	64	0m1.388s

Reverse word index

Extend the program with a `ReverseIndex` class. The `Map` function should produce for **each word** in the input a pair (word, offset), where offset is the **byte offset** in the input file. The `Reduce` function should output (word, [offset, offset, ...]), sorted by ascending offset.

This should require few lines of code (~25); if you find yourself writing much more code, you might be on the wrong track; ask for help to double check that you have the right plan. Fill in the `ReverseIndex` skeleton code in reverseindex.py.

While developing `ReverseIndex`, you want to use a small input file for which you know what the right answer is, so that you can quickly iterate to the correct solution.

Once you have a correct implementation, run it on the bible and look at the top 20 results. Your output should be as follows (without the top two entries "a" and "aaron"):

```
aaronites [1817624, 1875693]
abaddon [4789761]
abagtha [2165842]
abana [1643159]
abarim [732687, 767117, 767187, 944076]
abase [2304339, 2823950, 3322367, 3494485]
abased [3830632, 4056754, 4075715, 4589626]
abasing [4530190]
abated [28468, 29098, 29656, 573980, 950776, 1106814]
abba [3947502, 4412428, 4550068]
abda [1495474, 2149589]
abdeel [3121198]
abdi [1785594, 2005637, 2096175]
abdiel [1779568]
abdon [1048236, 1135107, 1135327, 1789463, 1797454, 1797941, 1804405, 2036728]
abednego [3468578, 3480066, 3482484, 3482731, 3482889, 3483421, 3483989, 3484242, 3484673,
3484736, 3485390, 3485499, 3485959, 3486358, 3486570]
abel [14223, 14233, 14448, 14563, 15039, 15141, 15228, 17635, 1222829, 1448386, 1448550,
1449216, 3834153, 4040933, 4686214, 4695492]
abelbethmaachah [1570895, 1700412]
```

Question 11: Once you are finished, submit your code in reverseindex.py on **gradescope** as a separate assignment. How long did it take you to complete this assignment?

---

11) About 4.5 hours.