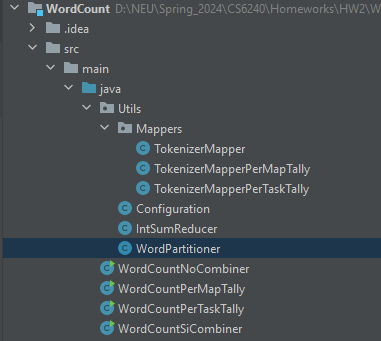
**Name:** Khoa Cu Dang (Cody) Cao

**Course:** CS6620 Spring 2024

# Homework 2 Report

1. **Souce Code**

For code reuse purposes, I separated components into their own classes (Mapper, Reducer, Partitioner) then have different main classes utilize them as needed.

**Common Utility Classes**

A computer screen shot of a program

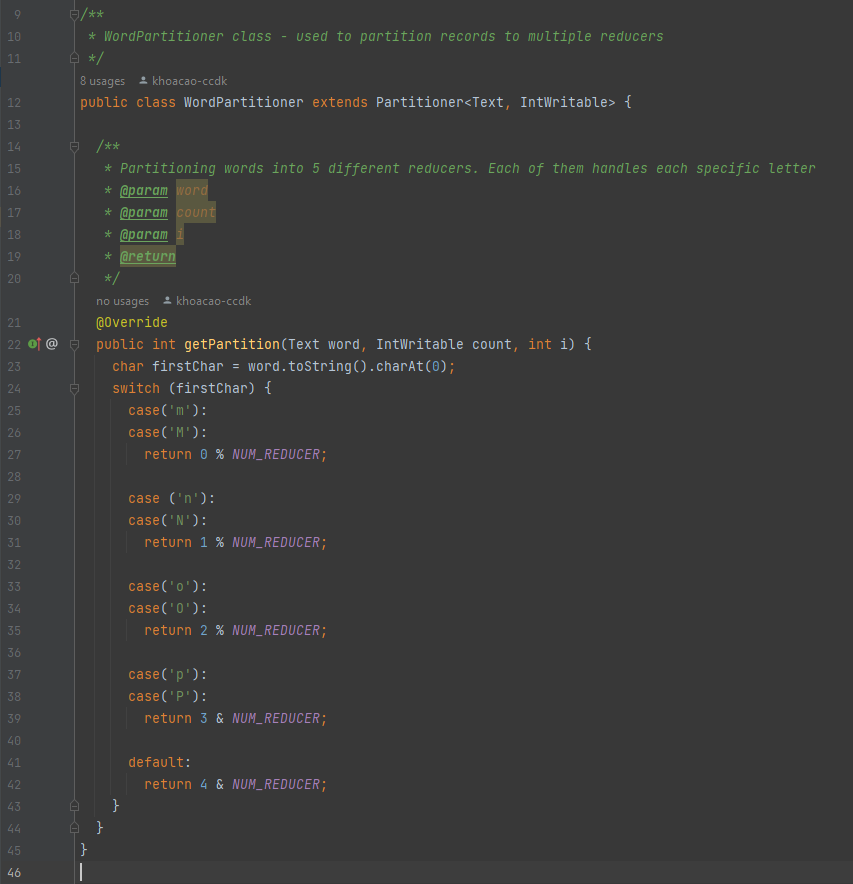
Description automatically generatedConfiguration: This class stores common configuration variables (list of acceptable starting characters, number of reducers being used)

IntSumReducer: This class is used for Reducer and in-mapper Combiner tasks.

A screen shot of a computer program

Description automatically generated

WordPartitioner: This class is used to distribute key-value pairs emitted by mappers to reducers based on the word’s starting character. Each reducer will handle a set of words that start with a specific character (both lowercase and uppercase).



**Mapper Classes**

There are three main parameters for the map() function within Mapper classes:

* Key – This parameter is often not used and/or set to null since in mapper phase, the focus is on processing the values. This variable is also not mentioned as being used by Hadoop documentation.
* Value – From the document of Hadoop, this Text value contains a line of the file. This then gets broken down further into separate words (separated by space) using StringTokenizer class. Thus, each call of the map() function will handle one line of the input file.
* Context: This object allows mappers to interact with the rest of the Hadoop system, as explained by a [StackOverflow post](https://stackoverflow.com/questions/26954162/what-is-keyword-context-in-hadoop-programming-world). This explains the context.write() function being used to emit records.

TokenizerMapper: This is the default implementation of the WordCount example given by Hadoop. I went ahead and added an additional condition that the words being processed must starts with a valid alphabetic character in order to be emitted.

A screen shot of a computer program

Description automatically generated

TokenizerMapperPerMapTally: With this per map tally implementation, the emitting record will be held off until the map has gone through every word in the input line. Aggregation is also performed using HashMap in case a word appears more than once in a single line.

A screenshot of a computer program

Description automatically generated

TokenizerMapperPerTaskTally: With this mapper implementation, I am having the aggregation at the mapper/input split level. Records are being held from emitting until the mapper has finished processing the whole input split. The records are aggregated using HashMap, then emitted using the cleanup() method of the mapper.

A screenshot of a computer program

Description automatically generated

**Job Classes**

WordCountNocombiner

A computer screen shot of a program code

Description automatically generated

WordCountSiCombiner

A computer screen shot of a program

Description automatically generated

WordCountPerMapTally

A computer screen shot of a program

Description automatically generated

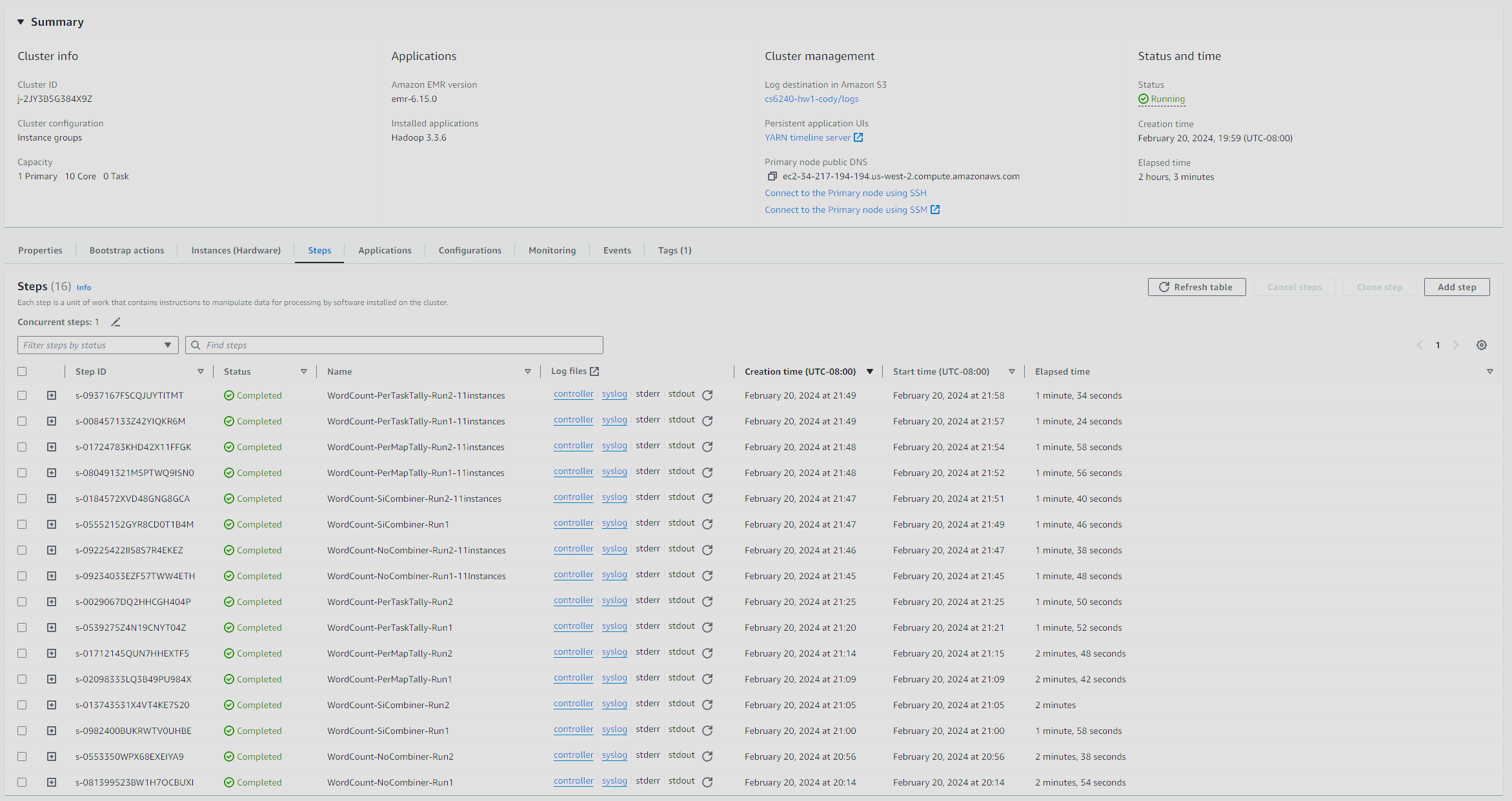
WordCountPerTaskTally

A computer screen shot of a program code

Description automatically generated

1. **Performance Comparison**

A white page with many lines

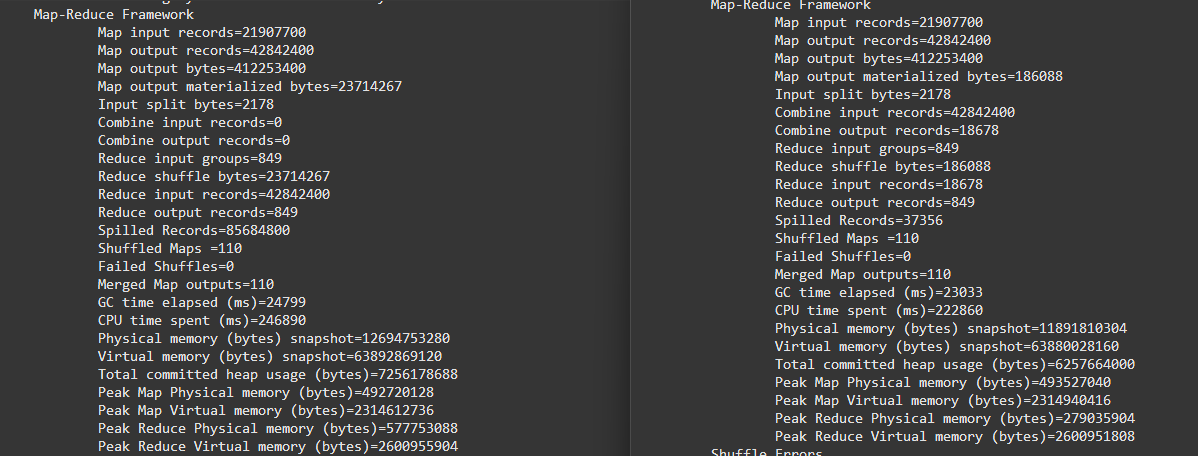
Description automatically generated with medium confidenceRecord of 16 Steps being run on AWS. After the first 8 steps, I edited the cluster configuration to 10 core nodes and 1 master node.

**A screenshot of a computer

Description automatically generated**

* **Do you believe the combiner was called at all in program SiCombiner?**
* **What difference did the use of a combiner make in SiCombiner compared to**

**NoCombiner?**

****

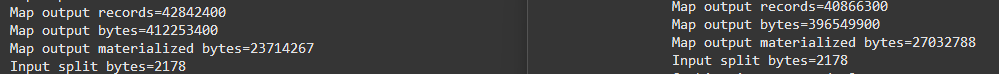
To answer both of the questions, we can look at the syslog for NoCombiner (left) and SiCombiner (right) run. When analyzing the syslog files, we can see that the combiner was definitely called in SiCombiner run due to:

* Combine input records (0 on NoCombiner vs 42842400 on SiCombiner)
* Combine output records (0 on NoCombiner vs 18678 on SiCombiner)
* Reduce input records (42842400 on No Combiner vs 18678 on SiCombiner)

This proves that the combiner was used in the middle of the Map phase and the Reduce phase, which reduced the number of input records on the Reduce phase significantly by doing aggregation before emitting records to the reduce phase. Additionally, having less input records means it took less time for the reduce tasks to run (321898ms on NoCombiner vs 175512ms on SiCombiner)



* **Was the local aggregation effective in PerMapTally compared to NoCombiner?**

****

Local aggregation in PerMapTally (right) seems to be a little bit better compared to NoCombiner (left)

* It has a slightly lower map output record (40866300 on PerMapTally vs 42842400 on NoCombiner)

This might be due to the fact that for each line, there is actually not many repetitive words. Thus, having a HashMap storing word count for only one line (PerMapTally) might not be an effective method when considering extra memory and additional runtime required to provision and operate an additional HashMap.

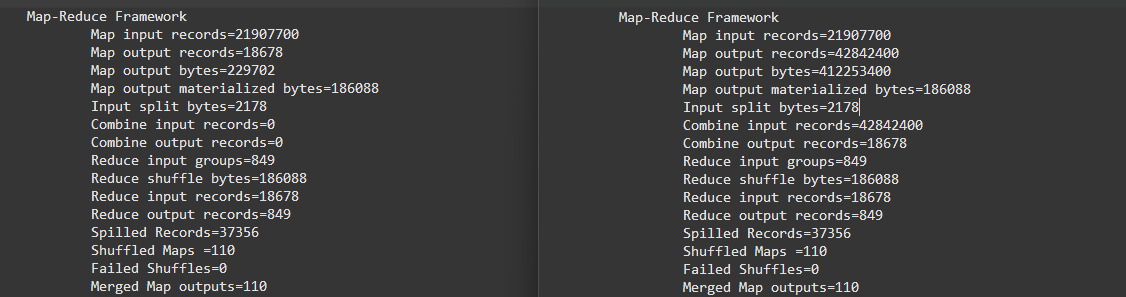
* **What differences do you see between PerMapTally and PerTaskTally? Try to**

**explain the reasons.**



When looking at the syslog of PerMapTally (left) vs PerTaskTally (right), there is a significant different in the Map out put record number (40866300 on PerMapTally vs 186088 on PerTaskTally), resulting in a significantly lower Reduce input records number. This can be because of the nature of the input file:

* Within a line of the input file, there is a fewer number of repetitive words. Thus, the HashMap provisioned in PerMapTally would have less value in aggregating the values.
* However, when we consider a split of the input file, there is a larger number of repetitive words. In this scenario, a HashMap provisioned at a mapper instance level (PerTaskTally) would help tremendously with aggregating records and reducing the number of output records.
* **Which one is better: SiCombiner or PerTaskTally? Briefly justify your answer.**



Both results, PerTaskTally (left) and SiCombiner (right), while having slightly different aggregation strategies, result in similar number of input records in the reducer phase. Both approaches also maximize the use of parallel computing by either having an in-mapper aggregation using a HashMap or using a custom combiner that aggregating the output records of the mappers records locally. Thus, I would consider them as equally good.

* **NEW: Comparing the results for Configurations 1 and 2, do you believe this MapReduce program scales well to larger clusters? Briefly justify your answer**

When looking as the total Step runtime from EC2, we can see that the steps that ran with 11 instances tend to be around 20 seconds faster compared to those that ran with 6 instances. This does show that there is a benefit to running more cluster. However, when looking at other factors (CPU time spent, total time spent by all mappers) there is actually less difference between the two configuration. Which means that the program does not benefit from having more instances, thus does not scale particularly well with a larger cluster.