COSC 6339 Homework1 Report

Chonghua Li

March 2017

Problem Description

In this assignment, we are given a collection of 216 books. Our goal is to use Hadoop MapReduce framework to 1). calculate the occurrence of all words in the provided books on a per book basis, and 2). determine how many books a word appears in. We will also compare the execution time of the code when using 2, 5 and 10 reducers.

Solution Strategy

Since we are using python to implement the applications for this assignment, we will use pydoop, the Python interface for Hadoop, to write our MapReduce applications. Pydoop offers a rich HDFS API and a MapReduce API, which allows to write pure python record readers/writers, partitioners and combiners. In order to do this, we need to import *pydoop.mapreduce.api* in the code.

Each MapReduce application has a Mapper class that contains a map function and a Reducer class that has a reduce function. The map function performs filtering and sorting, while the reduce function performs a summary operation. The default input format is TextInputFormat.

Based on the requirements of the application, we could also provide the optional classes, Combiner class and Partitioner class. The Combiner class is used to summarize the map output records for the same key, which would reduce the amount of data transfer between mappers and reducers. The Partitioner class is used to direct the output from the Mappers to the desired Reducers. The detailed solution strategy for each problem is described as follow.

1. **Counting the occurrence of all words on a per book basis**

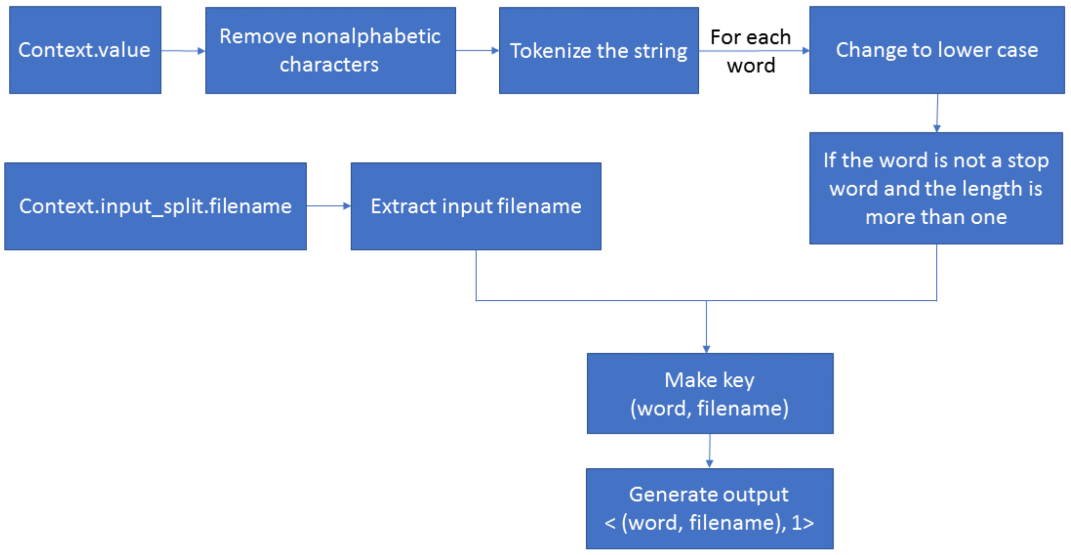
   Mapper Class

   For a simple word count application, the intermediate key-value output of the mapper will be <word, 1> for each word that the mapper encounters. However, in order to count the occurrence of each word on a per book basis, it's necessary to keep track of the book from which the word is coming. The key for solving this problem is to make a composite key that contains the word and the book name. The intermediate key-value pair output from the mapper in this case will be ***< (word, filename), 1>***.

   The filename of each input can be obtained from the context object that is passed into the user defined mapper by the Hadoop framework. The context contains a parameter called input_split, which returns the current input split as an InputSplit object. InputSplit represents the actual data chunk via filename, offset and length attributes. Hence, we can retrieve the file name of each input by using the statement: *os.path.basename(context.input_split.filename)*.

   The input value that we want to analyze can be obtained from context.value. It is first casted into string type and stripped off the nonalphabetic characters, followed by being tokenized by the python string split() function. All tokenized words were transformed into lower case to avoid unnecessary duplication. An additional step to screen for stop words was applied before making the intermediate key-value pairs outputs. Finally, the word and filename were made into a pair as a composite key, and the pair together with the count of 1 were emitted by calling context. emit(pair, 1). The diagram of the logical flow for mapper class is shown in Figure 3.

Figure 3. Diagram of the solution strategy for the mapper class in part 1.


Context.value
Remove nonalphabetic characters
Tokenize the string
For each word
Change to lower case
If the word is not a stop word and the length is more than one
Context.input_split.filename
Extract input filename
Make key (word, filename)
Generate output < (word, filename), 1>

Reducer class
The reducer class receives the output from the mapper, sum up the values from the same key, and write the result in the format of <(word, filename), sum> to the output file.

Partitioner class
The Partitioner class is an intermediate step between mapper and the reducer. It receives the key in the output from mapper and determine which reducer the data will be sent based on some user defined criteria. In order to arrange the final output entries that contain the same word in the same output file, the word from the mapper output key was extracted and mapped into an integer by using the python built-in hash() function. The id of reducer was determine by *reducer_id = (hash(key.split()[0])& sys.maxint)% num_reduces*.

2. **Determine how may books a word appears in**
The final outputs from part 1 contain records in the format of "(word, filename), count". We could easily get the number of books that a word appears in by analyzing the (word, filename) information in the output files from part1. In other words**, the number of times that a word occurs in the output file of part 1 = the number of books that the word appears in**.

The input data for this application is the output files from part1 with 5 reducers (hence there will be 5 files as input data).

Mapper class:
The input for the mapper class is in the format of "(word, filename), count", therefore we need to remove punctuations and extract the word. This is achieved by the following statement: *word=(str(context.value)).translate(None, string.punctuation).split()[0]*. The output of mapper is (word, 1) for each input it receives.

Reducer class:
The reducer simply sum up the total counts for a given word and emit the (word, count) as the final output.

Results

1. **Resources**
   The applications were compiled and run on the whale clusters: whale.cs.uh.edu.

   57 Appro 1522H nodes (whale-001 to whale-057)

   - two 2.2 GHz quad-core AMD Opteron processor (8 cores total)
   - 16 GB main memory
   - Gigabit Ehternet
   - 4xDDR InfiniBand HCAs (unused at the moment)

   Network Interconnect

   - 144 port 4xInfiniBand DDR Voltaire Grid Director ISR 2012 switch (donation from TOTAL, shared with crill)
   - two 48 port HP GE switch

   Storage

   - 4 TB NFS /home file system (shared with crill)
   - 7 TB HDFS file system (using triple replication)

2. **Part1 Results**
   The code was run with 2, 5, and 10 reducers for 5 times, respectively. The input data is located in HDFS *cosc6339_s17/books-longlist*. The performance metrics in terms of different execution time are summarized in table 1 and in Figure1.1 - 1.8.

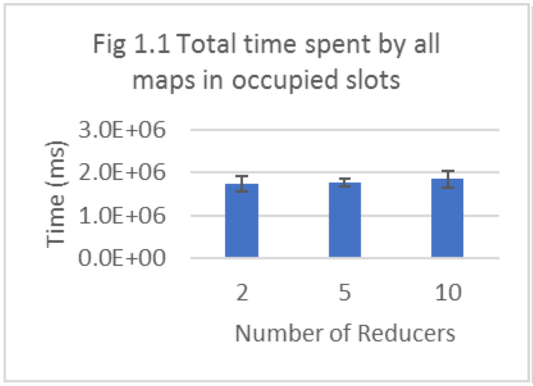
   From the result, we can see that the number of reducers has little impact on total time spent by all maps in occupied slots (Figure 1.1) and total megabyte-seconds taken by all map tasks (Figure 1.5). But it seems that increasing the number of reducers slightly correlates with the increase of the total time spent by all map tasks (Figure 1.3). On the other hand, the number of reducers positively correlated with all of time measurements on the reducer side (Figure 1.2, 1.4, 1.6). It also positively correlated with the CUP time spent (Figure 1.7).

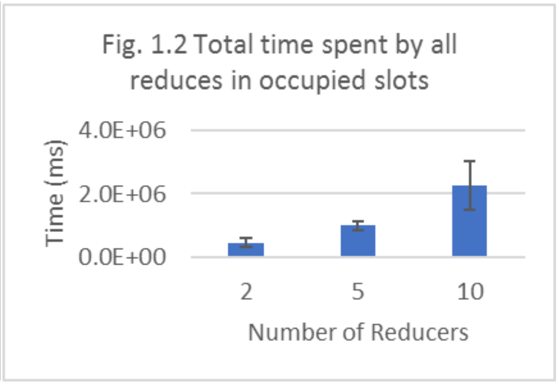
   Figure 1.8 indicates that 5 reducers setting gave the shortest average execution elapsed time, while 10 reducers setting gave the highest average execution elapsed time. However, since the standard deviations are relatively big, it may not be statistically significant.

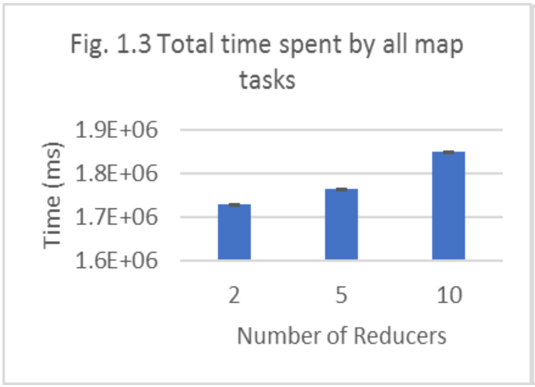
Table1. The summary part1 performance.

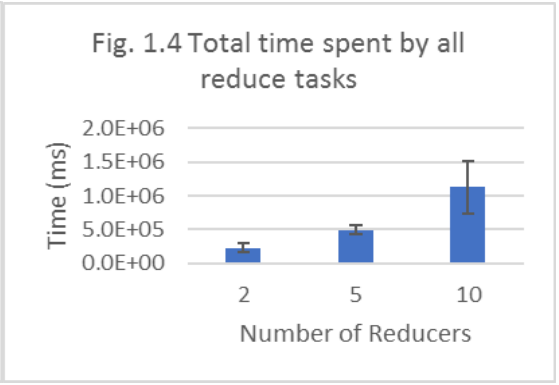
	2 Reducers			5 Reducers			10 Reducers		
	min	max	ave	min	max	ave	min	max	ave
Total time spent by all maps in occupied slots	1.52E+06	1.97E+06	**1.73E+06**	1.65E+06	1.89E+06	**1.76E+06**	1.62E+06	2.09E+06	**1.85E+06**
Total time spent by all reduces in occupied slots	3.15E+05	6.42E+05	**4.61E+05**	8.01E+05	1.14E+06	**9.89E+05**	1.38E+06	3.29E+06	**2.25E+06**
Total time spent by all map tasks	1.52E+06	1.97E+06	**1.73E+06**	1.65E+06	1.89E+06	**1.76E+06**	1.62E+06	2.09E+06	**1.85E+06**
Total time spent by all reduce tasks	1.58E+05	3.21E+05	**2.30E+05**	4.00E+05	5.72E+05	**4.94E+05**	6.92E+05	1.64E+06	**1.13E+06**
Total megabyte-milliseconds taken by all map tasks	6.24E+09	8.06E+09	**7.09E+09**	6.74E+09	7.76E+09	**7.23E+09**	6.65E+09	8.58E+09	**7.57E+09**
Total megabyte-milliseconds taken by all reduce tasks	1.29E+09	2.63E+09	**1.89E+09**	3.28E+09	4.69E+09	**4.05E+09**	5.67E+09	1.35E+10	**9.23E+09**
CPU time spent	3.40E+05	3.54E+05	**3.46E+05**	3.67E+05	3.79E+05	**3.73E+05**	3.92E+05	4.10E+05	**4.01E+05**
elapsed time (sec)	113	190	**155**	109	145	**127**	100	275	**204**

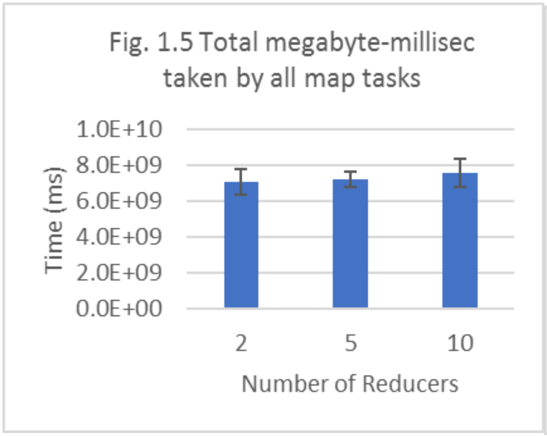
- Total time spent by all maps/reduces in occupied slots (ms) - total time map/reduce tasks were executing;
- Total time spent by all map/reduce tasks (ms) - wall-time resources were occupied by mappers/reducers;
- Total megabyte-seconds taken by all map/reduce tasks (ms) - Aggregated amount of memory (in megabytes) mappers/reducers have allocated times the number of seconds mappers/reducers have been running;
- CPU time spent (ms) - Cumulative CPU time for all tasks;
- Elapsed time (sec): total execution time from job accepted by the cluster to completion.

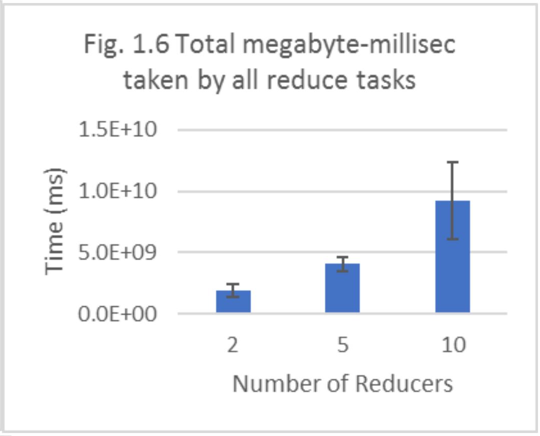


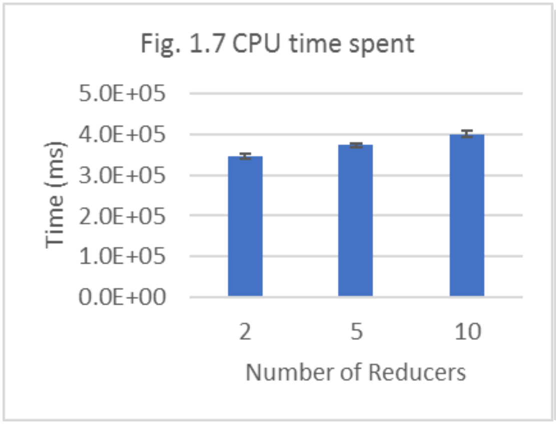


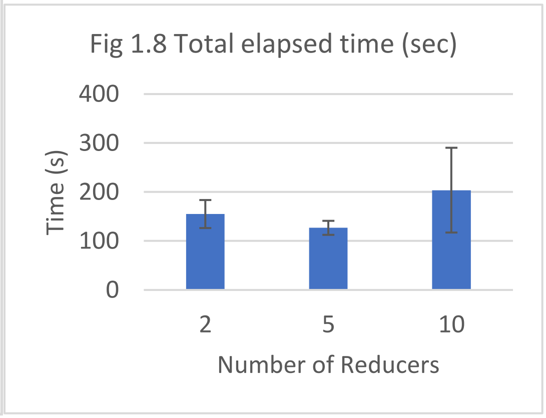












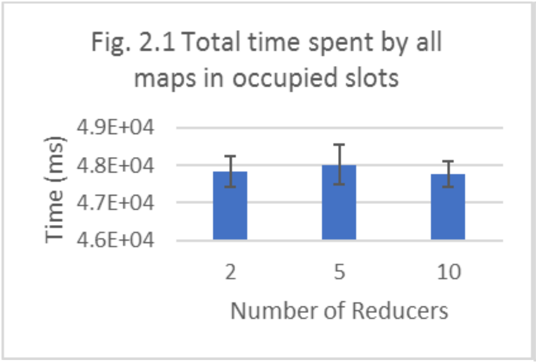
3. Part2 Results

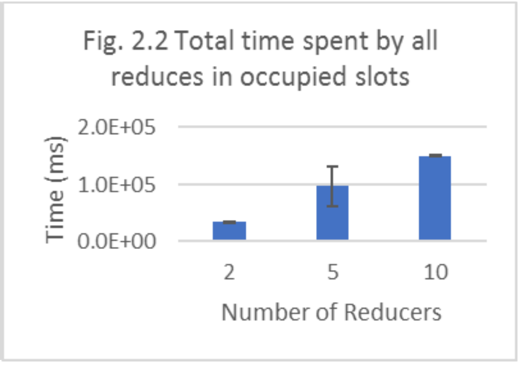
The code was run with 2, 5, and 10 reducers for 4 times, respectively. The input data is the output result of part 1 with 5 reducers (hence there are five input files). The performance metrics in terms of different execution time are summarized in table 2 and in Figure2.1 - 2.8.

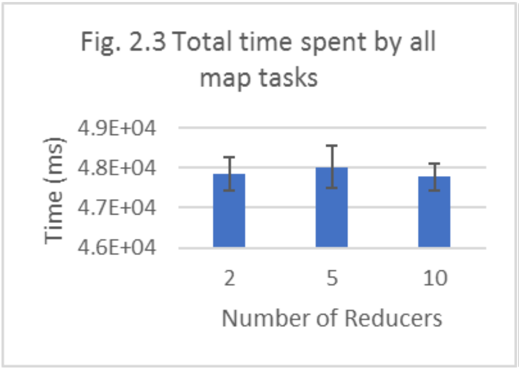
From Figure 2.1, 2.3 and 2.5, we can see that the reducer numbers have no effect on the time measurements on the map side. However, it positively correlates with the time measurements on the reduce side (Figure 2.2, 2.4, 2.4), as well as the CPU time spent (Figure 2.7). The average elapse time were about the same regardless the number of reducers used (Figure 2.8).

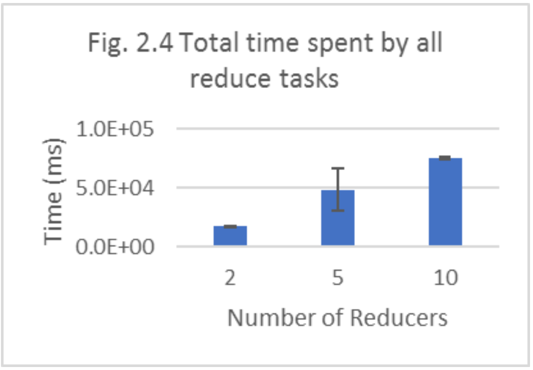
Table 2. The summary of part 2 performance.

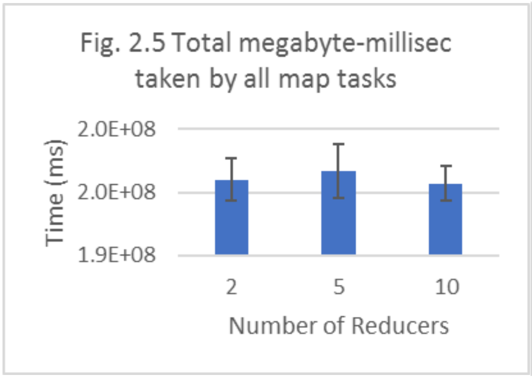
	2 Reducers			5 Reducers			10 Reducers		
	min	**max**	**ave**	**min**	**max**	**ave**	**min**	**max**	**ave**
Total time spent by all maps in occupied slots	4.74E+04	4.84E+04	**4.78E+04**	4.74E+04	4.86E+04	**4.80E+04**	4.74E+04	4.82E+04	**4.78E+04**
Total time spent by all reduces in occupied slots	3.42E+04	3.59E+04	**3.51E+04**	7.80E+04	1.50E+05	**9.70E+04**	1.48E+05	1.52E+05	**1.50E+05**
Total time spent by all map tasks	4.74E+04	4.84E+04	**4.78E+04**	4.74E+04	4.86E+04	**4.80E+04**	4.74E+04	4.82E+04	**4.78E+04**
Total time spent by all reduce tasks	1.71E+04	1.79E+04	**1.75E+04**	3.90E+04	7.51E+04	**4.85E+04**	7.42E+04	7.59E+04	**7.51E+04**
Total megabyte-milliseconds taken by all map tasks	1.94E+08	1.98E+08	**1.96E+08**	1.94E+08	1.99E+08	**1.97E+08**	1.94E+08	1.97E+08	**1.96E+08**
Total megabyte-milliseconds taken by all reduce tasks	1.40E+08	1.47E+08	**1.44E+08**	3.20E+08	6.15E+08	**3.97E+08**	6.08E+08	6.22E+08	**6.16E+08**
CPU time spent	3.15E+04	3.26E+04	**3.21E+04**	4.26E+04	4.29E+04	**4.27E+04**	5.92E+04	6.04E+04	**5.96E+04**
elapsed time (sec)	30	64	**39**	30	57	**38**	31	188	**88**

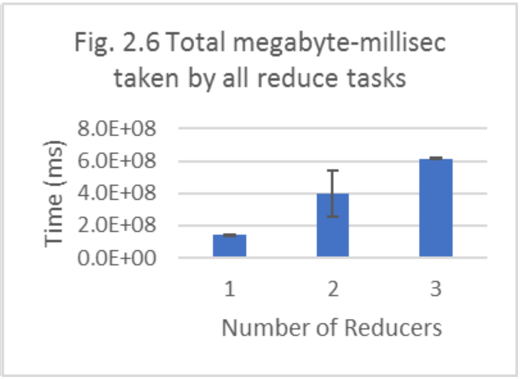


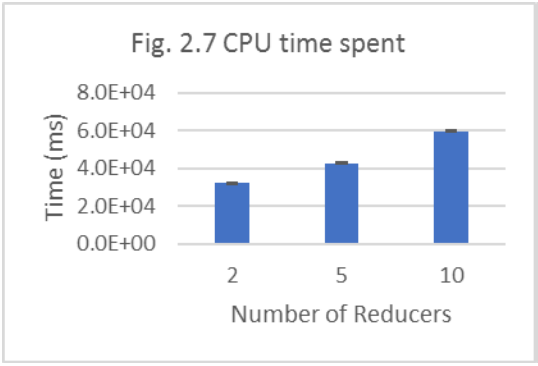


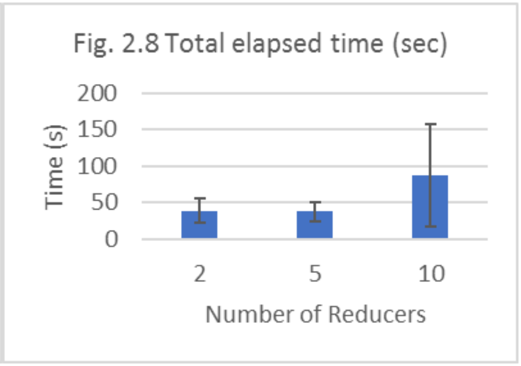












4. **The effect of user-defined Partitioner on the execution time**

In part 1, since the key in the intermediate output is a composite key made up by the word and the filename, the code was implemented with customized partitioner class to make sure that the records for the same word will be written to the same output file. In order to investigate the effect of the partitioner on the execution time for part 1, I removed the partitioner function and run the code on 5 reducers for 5 times. The elapsed time for execution obtained from both code is shown in table 3.

Table 3. The elapsed time for executing part 1 with or without customized partitioner.

	Min (sec)	Max (sec)	Average (sec) (n=5)
With partitioner	109	145	127
Without partitioner	47	105	74

Clearly, the default partitioner provided better performance in terms of execution time. However, the output generated by the code with customized partitioner is more organized.