Map Reduce

Czar Ian Echavez

xf008064

Contents

[Introduction 2](#_Toc507376507)

[High Level description 2](#_Toc507376508)

[Layout of the program 3](#_Toc507376509)

[Map Reduce Components (Low level description) 4](#_Toc507376510)

[Data Partitioner (Chunker) 4](#_Toc507376511)

[Data Chunk Class 4](#_Toc507376512)

[Mapper 4](#_Toc507376513)

[Combiner 5](#_Toc507376514)

[Reducer 5](#_Toc507376515)

[Error Checking 5](#_Toc507376516)

[Regex examples 6](#_Toc507376517)

[Flight number 6](#_Toc507376518)

[Passenger number 6](#_Toc507376519)

[Airport codes 6](#_Toc507376520)

[Airport name 6](#_Toc507376521)

[Longtitude and Latitude 6](#_Toc507376522)

[Output 6](#_Toc507376523)

[Flight Count 6](#_Toc507376524)

[Flight list 7](#_Toc507376525)

[Passenger Count 7](#_Toc507376526)

[Conclusion 8](#_Toc507376527)

[References 9](#_Toc507376528)

# Introduction

The files for the github project are here[[1]](#endnote-1): <https://github.com/Czar-Ec/MapReduce>

It should be the same as the folder attached to this submission.

This report details a program created to meet the requirements set by the Map Reduce assignment. The Map Reduce program needs to be able to parse 2 files as input, both of which have errors which need to be filtered out (instructions have been given as to what kinds of data are acceptable). The Map Reduce program will need to emulate a proper map reduce framework such as Hadoop or have a similar structure of a Map function and a Reduce function.

The report will be split into multiple parts; A high level description of the overall program will be described as well as how the project itself is stored as a repository. The report will also be describing the Map Reduce functions that are being replicated as well as an example of the data that the program will output. The report will also go through the function used to filter the input and how the program will handle the errors.

# High Level description

The map reducer I created is partially based on this map reducer[[2]](#endnote-2). The map reduce program almost follows the requirements set for this assignment, however is a little bit more complex as it also involves using the airport’s coordinates to calculate distances between airports in nautical miles.

The map reduce program that I referred to is also split into tasks, 3 of which are relevant to this assignment. The referred map reducer also has to determine flights from each airport, create a list of flights based on flight ID and to calculate the number of passengers for each flight. I have made my map reducer slightly different to the referred map reduce as I attempted to create my own implementation of a Map and Reduce function (the referred map reducer uses a TreeMap).

As stated before, the project is stored as a GitHub repository. This is so that I could work on the project on multiple machines and to make it easier to setup and get on with the work.

The program takes in 3 arguments when run on the command line. The 3 arguments are the passenger file, the airports file and the location of the output file (and its name) respectively. If no arguments are passed, the program is designed to load 2 test data (test1.csv and test2.csv which are essentially a copy of the AComp\_Passenger\_data.csv and Top\_30\_airports.csv, respectively) in the res folder and prints out the data to a file called outputFile.txt (also in the res folder). The program also aborts if neither of the files can be found so that the program does not break.

When the runnable jar is run from the file explorer, the map reduce defaults to running the program with no argument, where as when the runnable jar is run on the command line, the arguments can be provided; In windows command line the command to run the runnable jar is :

java -jar mapreduce.jar <file path for passenger> <file path for airport> <file path for output>

note: passenger and airport file needs to exist, but the output file is made at the position given with the file name given

## Layout of the program

The above shows a high level diagram of the overall program. The majority of the program processing is done just before and during the mapper portion of the program, this is because the mapper needs to be able to process the input data into key value pairs that the reducer can use as well as filtering erroneous data. Most of what the reducer does is just to sort the k3v3 pairs and turn them into a format that is printed onto the output file.

The data isn’t necessarily passed within the program but is instead printed out to files (in the res folder there are folders where the key value pairs are printed out and cleared once the program is run), this is to emulate the idea that multiple machines would be working on multiple data files with each running a thread and takes in an input file. Since the program is assumed to be able to process a large file, threading was used in order to be able to process files faster than when just using a single main thread. New threads are programmatically created for each chunk that is made by the program, since a chunk consists of 64 lines, then the number of threads would be:

Another thread is used in Combiner2; the thread is used to load all the k2v2 files in the k2v2 folder. One thread is assigned to each file so that all the k2v2 values are loaded into program memory much faster. (Note: There would be much less values for each file but loading the k2v2 files would be much faster if multiple files are being read into memory at the same time).

# Map Reduce Components (Low level description)

## Data Partitioner (Chunker)

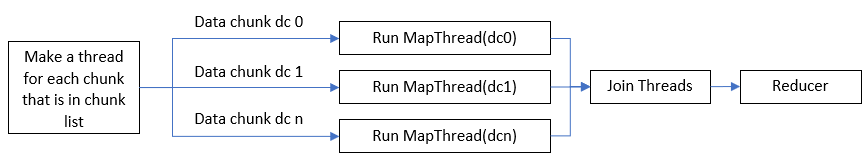
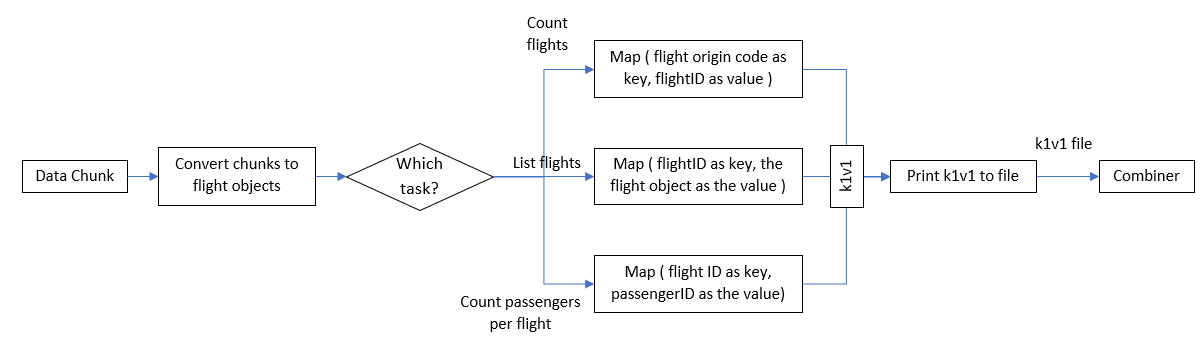
This component takes in the input file (just the passenger file since there are only ~30 values for airports). The Data Chunker creates a chunk which consists of a string containing the passenger data (at this point the string is not processed yet).

The chunking part of the program is where empty lines are removed. The Data Partitioner returns a list of chunks which the Mapper can then process.

### Data Chunk Class

The data chunk class is the object which acts as a chunk which stores a list of the string lines which will be processed in the mapper. The data chunk is given a maximum chunk size i.e. no more values can be added to the chunk once the list size is equal to the maximum chunk size. Once the data chunk returns to the Partitioner that the chunk is full, the data chunk is added to the partitioner’s chunk list and a new data chunk object is created.

## Mapper

Above is the general structure of the mapper. A thread is created and run for each data chunk in the data chunk list. Once all threads are created, the program then joins all the threads before continuing. The MapThread is given a specific data chunk to process and a task value so that the proper map process is done (there are 3 tasks which require different type of mapping).

The next diagram (directly above this) shows the structure of the MapThread. All Threads regardless of task will convert the chunks into flight objects (when the program is run, all the errors will be listed and this is done 3 times due to there being 3 tasks). The MapThread then diverges from different tasks since each task requires a different kind of mapping. Task 1 specifies that the program should list the number of flights FROM each airport, therefore the flights are mapped with the origin code as the key and the unique flight ID as the value. Likewise with all the other tasks, the mapping is done differently

* The flights need to be listed with all values, so the key would be the flight ID and the entire flight object is the value
* The passengers need to be counted for each flight, so the key would be the flight ID and the passenger ID is the value

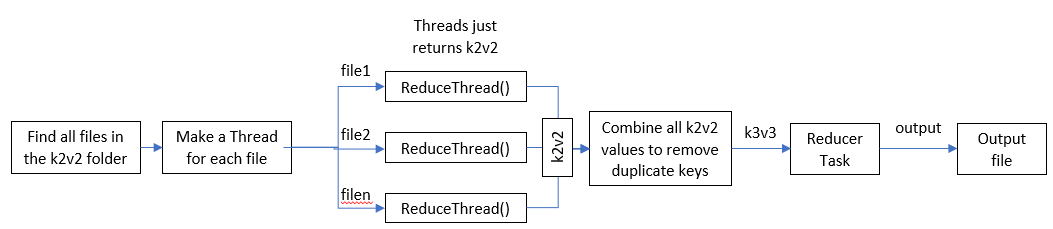
The k1v1 values created are then printed out (for reference, although in the program this is bugged and does not actually). The k1v1 are then processed by the combiner to create k2v2 pairs

## Combiner

The combiner is called within the MapThread i.e. there is a combiner run for each chunk of data. All the combiner does is take the k1v1 pairs and converts them into k2v2, which is removing the k1v1 values with the same keys and grouping all values with the same keys into the same k2v2.

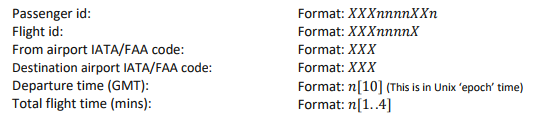
k2v2 pairs are basically a key value pair which has all the list of the values of the k1v1 with similar keys i.e. k2v2 = (<k1> <list<v1>>). These k2v2 pairs are then printed to a file in the k2v2 folder relevant to the current task

## Reducer

Above is the structure for the Reducer part of the program. The reducer will check if any files are created in the k2v2 folder relevant to the task and will try to load them into memory. A thread is made for each file and the reduce thread will return a k2v2 list. The k2v2 list will be processed again to create a k3v3 pair which is the same as the conversion of k1v1 to k2v2 but instead all the list values of one k2v2 is added to another.

The reducer task in the diagram is an abstracted process. Each task will have a different process for the reducer task. Once the task is done, the processed data is then printed to the output file.

# Error Checking

The assignment specifies that there is a specific format for some data types.

The program has a function that validates input called “validation” that takes 2 parameters; the string to be validated and the type of validation. The validation is done via the use of a regex; if the string does not match the regex, the entire line is not considered valid and so is not processed (the program prints the offending string and specifies which part of the string is invalid).

## Regex examples

### Flight number

The flight number consists of an alphanumeric value of length 8 but the first 3 and last value are letters and the rest are numbers. The regex used to check this is: [A-Z]{3}[0-9]{4}[A-Z]{1}

### Passenger number

The passenger number is also an alphanumeric value but its length is 10 and is in an XXXnnnnXXn format where X are letters and n are numbers. The regex used to check this is:

[A-Z]{3}[0-9]{4}[A-Z]{2}[0-9]{1}

### Airport codes

The airport codes are an alphabetical value that is 3 letters long; the regex check is simply: [A-Z]{3}

### Airport name

The airport name is simply any alphanumeric value that has whitespaces but has to be between 3 and 20 characters long. The check for this is to simply check if the length of the string is more than or equal to 3 AND less than or equal to 20.

### Longitude and Latitude

The longitude and latitude is a float value that can either be positive or negative but has between 1 to 3 numerical length before the decimal and is always 6 numbers long. The regex check is:

[+-]?\\d+{3}\\.?\\d+{6}\\s\*

# Output

All task outputs are printed on the same file that the user specifies. Each task is clearly separated by a line of “/////”. The title for the task is also placed to make it easier to look at the file. I suggest using notepad++ to read the output file since notepad does not display the new lines that should be output. (Notepad makes it difficult to look at the outputs)

-----------------------

Unmatched airports:

LAX

FRA

HKG

DXB

SIN

SFO

PHX

IST

-----------------------

## Flight Count

Flight count:

ATL: 34

PEK: 10

LHR: 24

…

On the left is an example of the output for the flight count. The airport code is listed alongside the number of flights the program has counted. The program also lists all the valid airports that did not have a matched airport (airports that did not have any flights depart from it)

## Flight list

Flight List:

---- XXQ4064B ----

Passenger ID: UES9151GS1

Flight ID : XXQ4064B

Origin: FRA

Destination: JFK

Departure Time: 17:05:17

Flight Length: 13 hours 22 minutes

Passenger ID: HCA3158QA3

Flight ID : XXQ4064B

Origin: FRA

Destination: JFK

Departure Time: 17:05:17

Flight Length: 13 hours 22 minutes

On the left is an example of the output for the flight list task. A sub heading is made for each flight and underneath is the flight values printed out along with the labels.

There are some duplicates which can be easily fixed by changing the mapping to remove duplicates, however due to time restrictions, I have decided to leave the program as it is.

Passenger Count:

XXQ4064B: 24

SOH3431A: 17

PME8178S: 17

MOO1786A: 12

## Passenger Count

On the left is an example of the final task. The flight ID is printed and next to it is the number of passengers the program has managed to count.

# Conclusion

Overall, I’m glad that the program is able to satisfy the specifications put forward by the assignment, however I am very certain that it can be improved. As stated in the output for the flight list, there are some duplicate data which could have been removed via the use of HashSets. The hash sets would be helpful in the removal of duplicate data.

There are some other problems in the program where some arguments are not being considered by the program and therefore the program always deletes all the outputs for the k1v1 and k2v2 in the res folder. If you would like to see the k1v1 and k2v2 outputs, you can request for me to edit the program so that the values are not deleted after the program runs.

As for the assignment itself, the research for a map reduce system was difficult as most of the results given were specifically only for the use of the Hadoop framework. I had used multiple resources, mainly the program I referred to2 and the tutorials point[[3]](#endnote-3) map reduce tutorials. As I didn’t want to copy the referred program exactly, I instead opted for making my own implementation of the Map function instead of using the TreeMap in built function used by the referred program.

# References

1. My Map Reduce implementation

   <https://github.com/cilliand/CloudComputingCoursework-MSc> [↑](#endnote-ref-1)
2. Map Reduce implementation from another person

   <https://github.com/cilliand/CloudComputingCoursework-MSc> [↑](#endnote-ref-2)
3. Tutorials point Map Reduce

   <https://www.tutorialspoint.com/map_reduce/index.htm> [↑](#endnote-ref-3)