

Develop a Software Prototype of a MapReduce-like system

Finley Williams – 23013019
F.Williams@student.reading.ac.uk

Contents

Abstract.....	2
Introduction.....	2
Task & Objectives.....	2
High-level description of the development of the software prototype.....	2
Description of Subversion command line process undertaken	4
Description of MapReduce functions replicated.....	4
Mapper, map().....	4
Shuffle.....	4
Sort.....	5
Reducer, reduce()	5
Output format of each Job.....	6
Objective One	6
Objective Two.....	6
Objective Three.....	6
Error Logs	7
Error handling strategy	7
Self-appraisal of MapReduce equivalent software	8

Abstract

Data analysis is a huge part of research for businesses and other institutes however as data sets get larger, standard data analysis techniques become less effective. Parallel processing is one such solution to this dataset size problem which we explore within this document.

A non-distributed non-MapReduce version of Apache's Hadoop implementation of MapReduce is created and a high level description of the development occurs. Error detection and error correction on the input data set occurs and discussion is had. Finally, the differences between this Apache's Hadoop and the developed implementation are compared and contrasted.

Introduction

As data sets get bigger they become more difficult to get useful information from. This problem is only getting worse as we create more data. Humans are incapable of processing the huge amount of data we create.

For this reason we have created methods of systematically analysing data. However as these data sets become even larger it has become impossible for a single computer to reasonably analyse the data set.

To solve this problem a method of analysing these huge data sets was developed. These methods often rely on distributed parallel computing. Spreading the load of the analysis between multiple machines enables the data-sets to be efficiently processed, stored.

Task & Objectives

This coursework's task is to create "non-MapReduce executable prototype" that is capable of completing the three objectives given:

- Determine the number of flights from each airport (i.e. Departures); include a list of any airports not used.
- Create a list of flights based on the Flight ID, the output should also include the passenger IDs of all passengers on board the flight, as well as the IATA/FAA codes, the departure time, the arrival time (in HH:MM:SS format) and the flight duration.
- Calculate the number of passengers on each flight.

High-level description of the development of the software prototype.

A high level diagram of the program flow is shown below in figure 1. This helps illustrate what I discuss later in this section.

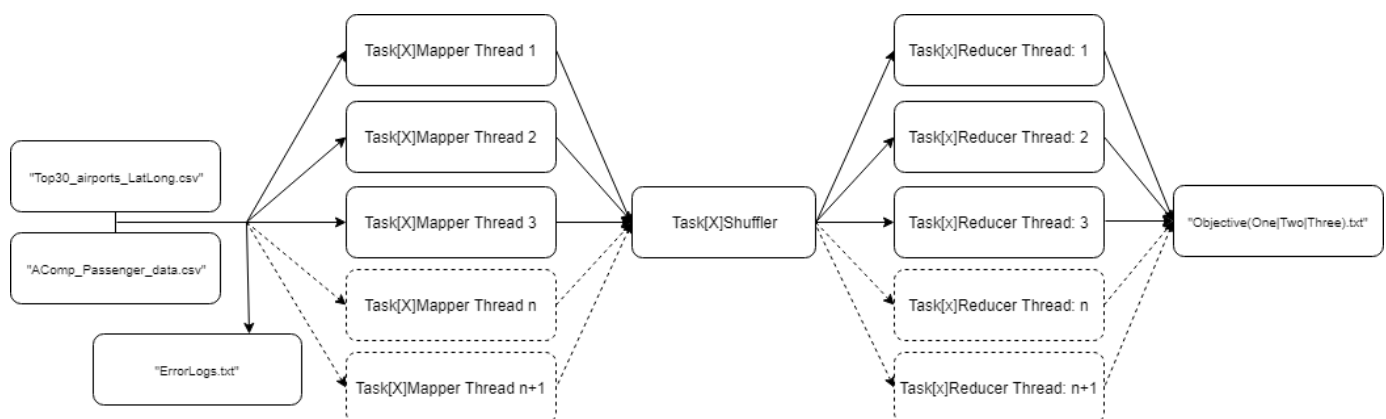


Figure 1: Program flow

A prerequisite for the task to be completed is to read in "AComp_Passenger_data.csv" to get the data required for the analytics. This required using a BufferedReader to take a line of the .csv and then

splits the line on each of the commas in the line. Error checking is then done on all fields to make sure that the inputted data is valid. This error checking is as per the assignment brief.

The same is done to “Top30_airports_LatLong.csv” to get a list of the airport codes and their names. Error checking is also done here to check if these airport names and codes are of the correct length and contain the correct characters.

All errors detected in either “AComp_Passenger_data.csv” or “Top30_airports_LatLong.csv” are written to “ErrorLogs.txt” so the errors can be checked.

The first step after this was to complete a full MapReduce (Map, Shuffle and Reduce) for a single task. The task chosen for this was task one. The decision to only implement one full MapReduce initially was made as once one version has been made the other two MapReduces can be made easily from the initial MapReduce with only small changes.

The first subtask of the above task was to create an object to hold the key value pair for this task. This was creating an object which when its constructor is called and is passed a row of data will assign the correct data to be its key and the rest of the data to be its values.

The next subtask would be create the mapper. This required the creation of a new class which was runnable as a Thread. The constructor of this is passed a single row of the data taken from the .csv. Within this class the object calls the constructor of the object created in the previous step passing it the row of data. This creates a new object which is the Key-Value pair for the task.

The next subtask is iterating over the data and creating a new mapper thread for each of the rows of data. This requires a loop which creates new mapper threads, passes them a row of data and then adds them to a list of mapper threads. Another loop is then required to iterate over the list of mapper threads to sequentially join and get the key value pair from the threads. These key-value pairs are then added into a list to allow the next step to be completed.

The next subtask is to create the shuffler, this requires the key-value pairs generated in previous step as a large list. This step aims to go from a list of key-value pairs to a list of key-list<values>. To do this the key-values are added to a hashmap. This is ideal as hashmaps do not allow duplicate keys. This is useful as if a duplicate key is detected, the values for that key can be loaded and the other values for that key can be added to the list and then written to the hashmap. This overwrites the previous value for the same key which has less values.

The final subtask is to create the reducer. This task is of similar structure to the creation of the mapper threads. As it also requires two loops to create and join the threads. The key difference being that the loop iterates over the key-list<values> creating a new reducer thread passing it a key-list<values> as a constructor and then adding it to a array of reducer threads. To complete task one the reducer must also be passed an array of airport code and names to use in the output. Finally another loop is created which loops over the list of threads sequentially joining them.

This loop also gets the required output data from these threads which is a string of the airport flown from for this reducer thread and the output string of how many flights from this threads airport. This output string then written to the file “ObjectiveOne.txt”. The string containing the airports is added to a list of airports which were flown from, from within the data, this is used to calculate the last part of task one.

The last part of task one is that it asks for a list of airports which were not flown from. This is calculated using the list of all airports which was read from the .csv in the beginning and the list of airports which was returned from the task one reducers. The two lists are compared and any airports which are not in the list from the reducers is then amended to the bottom of the output file, “ObjectiveOne.txt”.

Once task one has been made, I modified where necessary changing the key to flightID as this is the key for task two and three. The other changes made for task two and three is rebuilding the reducers for task two and three to return different outputs. A new mapper was not required for task three as it uses the same key as task two which allows it to use task two's mapper.

Description of Subversion command line process undertaken

GitHub was used to do subversion control of my development and then imported to GitLab under mb013019 as per the assignment requirements.

The repository can be found: <https://csgitlab.reading.ac.uk/mb013019/AdvCompCoursework>

The commands used were as follows:

To see what was changed:	git status
To add all changes to the commit:	git add .
To commit these changes:	git commit -m "The comment for this commit here"
To push the changes to the repository:	git push

Commits usually occurred when either a new feature had been finished or any bug had been fixed. The ability to revert to backup was extremely useful allowing reversion when a change had not worked as planned.

Description of MapReduce functions replicated

Mapper, map()

Apache Hadoop's Mapper

Hadoop's map function accepts a key and a value. From this it returns a Key-Value pair. One instance of the map function returns a single key-value pair. Multiple instances of map() are called to turn a data set into a set of key-value pairs. The important part here is that the multiple instances of map() are all running simultaneously and are merged into a list upon thread completion.

Implemented Mapper and it's differences to Hadoop's

Hadoop's map() function has been replicated by the functions, Task1Mapper and Task2Mapper. These functions act as map() as they accept a key and a value and return a Key-Value pair.

Within each of the mapper functions a new object, Task[X]JobObject is created. This Object is passed; the Key as a string and the Values as an array of strings. This Object is then added to the ArrayList to be returned when the Thread is finished. This ArrayList of JobObjects is then returned to the main() and combined into a single ArrayList of every key-value pair for a given data-set.

Much like Hadoop's MapReduce, a instances of the Mapper is created for every Key and Value passed.

This function extends Thread and overrides run(). This allows the function to be called as a new Thread which allows for the parallel processing. This ability to run in a parallel method is similar to Hadoop's map() with the key difference in that it is not distributed and cannot be run over HTTPS.

Shuffle

Apache Hadoop's Shuffler

Apache Hadoop's shuffler is actually the first phase of the reducer. However when the reducer is called the shuffler runs and copies all the outputs from the mappers using HTTPS from the distributed network.

Implemented Shuffler and it's differences to Hadoop's

Hadoop's shuffle has been replicated by the functions Task1Shuffler and Task2Shuffler.

These are passed the ArrayList of Key-Value pairs from the appropriate mapper, this is similar to Hadoop's shuffle function.

The shuffler then iterates over the list of List of Key-Value pairs adding each to a HashMap as a Key and an ArrayList. When it finds a Key which already has a Value associated it loads the Value and adds the value to its ArrayList so that a Key can have multiple values.

When the shuffler has iterated over the entire ArrayList a HashMap is returned to the main(). This returned HashMap is actually a Key-List<Value>. This is also what is outputted by Hadoop's MapReduce.

Unlike Hadoop's shuffle function, Task1Shuffler and Task2Shuffler run in between the Mapper and the Reducer as a separate stage. This is different to Hadoop's MapReduce as Hadoop runs the shuffler in the first phase of the reducer, whereas this implementation runs the shuffler before the reducing starts.

Another key difference between this implementation of shuffle, and Hadoop's implementation is that this implementation contains the sort function which creates list of values turning the Key-value pairs into key-list<values>. This is further explained in the Sort section below.

Sort

Apache Hadoop's Sort

Much like Apache's shuffler the sort is actually a phase of the reducer.

When the reducer is called and the shuffler is running, the sorter takes the key-values copied by shuffler and groups the Key-Value pairs into Key-List<Value>.

The important part here is that this sort runs simultaneously to the shuffle aggregating values as the shuffler pulls them in.

Implemented Sort and it's differences to Hadoop's

Hadoop's sort has been replicated by the functions Task1Shuffler and Task2Shuffler as part of the shuffle function for this implementation.

The sorting part of the two shuffle functions is run after they've been collated by the shuffler. All duplicates are removed as the HashMap only allows Unique keys meaning if a Key is found the List of values is loaded and the additional value is appended to the list. This is then re-added to the list under the same key. This results in a Key-List<Value> list which is returned to the main().

Reducer, reduce()

Apache Hadoop's reduce()

As mentioned in the previous two sections Apache Hadoops reducer first runs the shuffler and the sort. These mean that the reducer is passed Key-Value Pairs. The shuffler and sort turn this into a Key-List<Value>. This Key-List<value> is what is used to generate the data. The reducer runs for each key returning both the key it operated on and the output. Hadoop's implementation returns the key it operated on as due to its distributed nature it would not know which output was which if it was not given the key.

Implemented reduce and it's differences to Hadoop's

Hadoop's reduce feature has been replicated by the functions Task1Reducer, Task2Reducer and Task3Reducer.

Task3Reducer accepts the key as a string, an ArrayList<String[]> which contains the values associated with the key.

Task1Reducer and Task2Reducer also accepts a two dimensional array of the list of all airport codes and all airport names as well as the previously mentioned key and values.

All three of my reducer return a string to the main() which is then written to the output file. It does not return a key as the main thread knows which output is from which key.

Output format of each Job

The output for the three MapReduces can be found under “ObjectiveOne.txt”, “ObjectiveTwo.txt” and “ObjectiveThree.txt”. Where possible they have been tabulated/spaced for maximum readability.

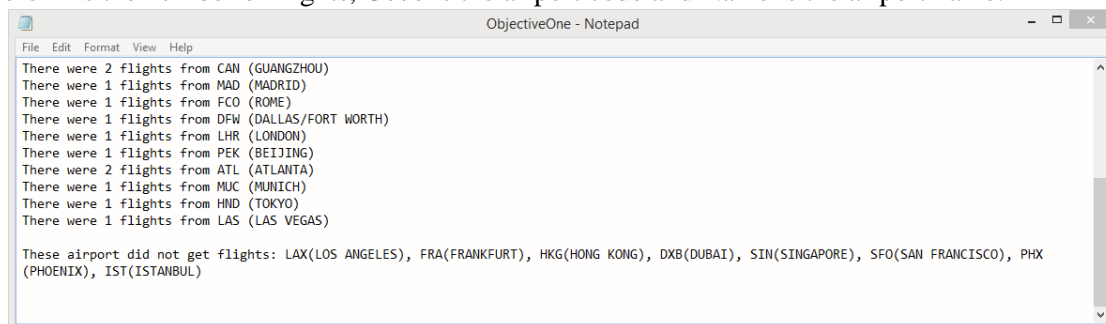
“ErrorLogs.txt” is also included as this is an output of the program getting the data from the two previously mentioned .csv’s

Objective One

ObjectiveOne.txt lists each airport and its number of flights in form:

There were X flights from CODE (NAME).

Where X is the number of flights, Code is the airport code and Name is the airport name.



```
ObjectiveOne - Notepad
File Edit Format View Help
There were 2 flights from CAN (GUANGZHOU)
There were 1 flights from MAD (MADRID)
There were 1 flights from FCO (ROME)
There were 1 flights from DFW (DALLAS/FORT WORTH)
There were 1 flights from LHR (LONDON)
There were 1 flights from PEK (BEIJING)
There were 2 flights from ATL (ATLANTA)
There were 1 flights from MUC (MUNICH)
There were 1 flights from HND (TOKYO)
There were 1 flights from LAS (LAS VEGAS)

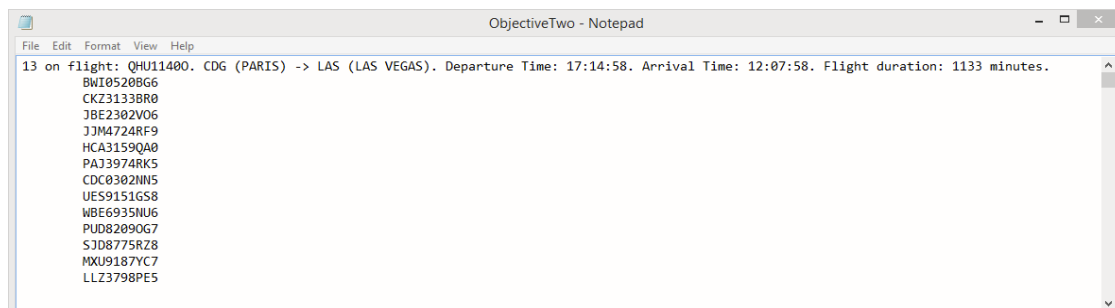
These airport did not get flights: LAX(LOS ANGELES), FRA(FRANKFURT), HKG(HONG KONG), DXB(DUBAI), SIN(SINGAPORE), SFO(SAN FRANCISCO), PHX (PHOENIX), IST(ISTANBUL)
```

Figure 2: ObjectiveOne.txt

Objective Two

ObjectiveTwo.txt lists the number of people on a flight, the flight code, the departure airport code and name, the arrival airport code and name, the departure and the arrival time, both in HH:MM:SS and the flight duration.

Listed below this is the PassengerID of each passenger on this flight.



```
ObjectiveTwo - Notepad
File Edit Format View Help
13 on flight: QHJ11400. CDG (PARIS) -> LAS (LAS VEGAS). Departure Time: 17:14:58. Arrival Time: 12:07:58. Flight duration: 1133 minutes.
BWI0520BG6
CKZ3133BR0
JBE2302V06
JJM4724RF9
HCA3159QA0
PAJ3974RK5
CDC0302NN5
UES9151GS8
WBE6935NU6
PUD8209OG7
SJD8775RZ8
MXU9187YC7
LLZ3798PE5
```

Figure 3: ObjectiveTwo.txt

Objective Three

ObjectiveThree.txt displays the number of people on each flightID. It displays this in the format: X passengers on flightID --> ID, where X is number of passenger and ID is the flightID of that row.

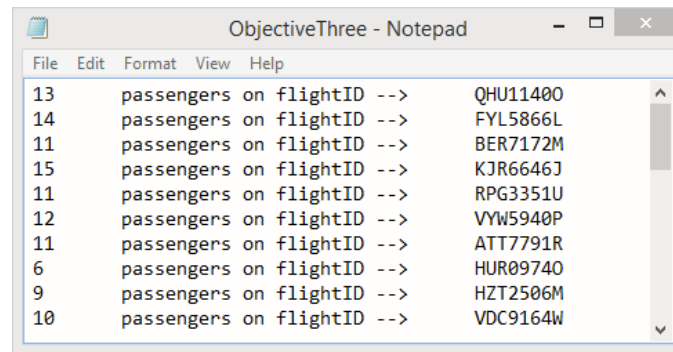


Figure 4: ObjectiveThree.txt

Error Logs

Each row describes the type of error found, where applicable displays the erroneous data and finally says what it will do to the row that the error is on.

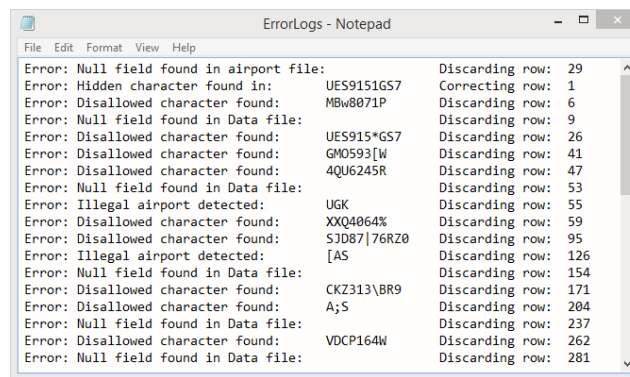


Figure 5: ErrorLogs.txt

Error handling strategy

All errors are contained within “ErrorLogs.txt”. This allows all errors to be reviewed and verified.

During the reading of data from “AComp_Passenger_data.csv” and “Top30_airports_LatLong.csv” the data is checked for validity against the requirements given in the assignment brief.

The first type of error detection the data goes through is making sure data is actually there. This is achieved by checking the all of the six columns do not equal “”. If any are found to be empty the row is discarded and an error is written to “ErrorLogs.txt” an example of this error type for both the airport file and the data file is shown below:

“Error: Null field found in airport file: Discarding row: 29
Error: Null field found in Data file: Discarding row: 9”

The next error detection run is against the requirements found within the assignment brief. These requirements are shown below:

PassengerID : XXXnnnnnXXn
FlightID : XXXnnnnnX
Airport : XXX
Departure time : n[10]
Airport name : (X\s)[3. .20]
Airport : XXX
Total flight time : n[1. .4]

In this format, X is an uppercase digit and n is an integer number, if n is followed by square brackets the character following the n denotes the number of characters allowed of this type. These requirements are checked character by character sequentially along the strings.

This error detection revealed multiple errors of this type and one false positive error of a different type. An example below is shown of a correct error found by this validation method

“Error: Disallowed character found: XXQ4064% Discarding row: 59
Error: Disallowed character found: ~AN, Discarding row: 461”

This method of validation also uncovered a false positive as the very first character in the file is “u”\uffeff”. This is a start of file character used in UTF-16 BOM which is used to tell the difference between little-endian and big-endian encoding.

Other than this character the row of data is completely correct. This is a false positive so steps were taken to correct this and any similar errors. All string are now checked for invisible unprintable characters before other validation is run. If any invisible characters are found the character is removed and the error corrected. Fortunately no other errors of this type are detected in the data. An example of this type can be seen below:

“Error: Hidden character found in: UES9151GS7 Correcting row: 1”

The final type of error correction ran was checking that all syntactically correct airport names were actually included in the airport file. If the airport is syntactically valid but not a real airport an error will be displayed and the row rejected. An example of this error type can be shown below:

“Error: Illegal airport detected: yEK Discarding row: 494”

This covers all forty errors and one error correction made over both “AComp_Passenger_data.csv” and “Top30_airports_LatLong.csv”.

Self-appraisal of MapReduce equivalent software

Overall, this reproduction of Apache Hadoop’s MapReduce has gone very well. The structure is reasonably well mimicked with exception of the shuffle and sort being its own function as opposed to being part of the reducer. Each mapper and reduce thread is passed a single set of data keeping the one thread, one data set nature similar.

If this task were to be repeated one aim to improve on would be to move the shuffle and sort function to the reduce function.

Other improvement to be made could be more error checking such as seeing if passengers are on multiple flights at the same time or checking if all the passengers on a given flights arrival and departure time match up, this could also include an element of error correction as if a passengers flights time did not match but every other passengers did it would not be difficult to correct the erroneous time.

A design improvement to be made could be to create a “generic” jobobject and a “generic” mapper as currently these are practically the same code wise, with minor improvement I could reuse the class.

A different improvement to be made could to attempt to match rows with single incorrect characters again the rows in the data set. If only a single character in the row is wrong and everything else is correct it would be safe to assume a correction can be made.

A final improvement to be made could potentially be adding the ability to run this non-MapReduce as a distributed system over HTTPS for improved parallel processing.