CSMCC16 - Cloud Computing - Develop a software prototype of a MapReduce-like system

Passenger Airline Flights; Student ID: 24885531

[Introduction](#_2fivto2xsu8)

[High-Level description of the development](#_opthdidhwou5)

[Reproducing the results](#_jyg2lshqok7k)

[Installing Julia](#_kc3kjqdrqgoh)

[Installing the NMR package](#_1iv2sxrxs4gb)

[Running the objectives](#_xd8pe85znrej)

[Simple description of the Version Control command line process undertaken](#_nl65vlz6se15)

[Detailed description of the implemented MapReduce functions](#_eu1f5cqmqowx)

[Objective #1](#_jo03w1c3k3aq)

[Objective #2](#_d3mvc6x7vr4k)

[Objective #3](#_olxhhfnlj9k1)

[Objective #4](#_ss9ev6mm7t53)

[Input data error detection and correction](#_sw8r882fzvba)

[Lessons learned](#_bwky9lsst23p)

[Self appraisal](#_2fwp7fav8eq)

[References](#_3rl9wigcxmcm)

# Introduction

The objective of this assignment is to develop a non-MapReduce executable prototype that follows the MapReduce paradigm but without using any readily available implementations. Once this is done, four reports need to be implemented using the prototype. Each of the four reports uses provided Flight data to generate its results.

# High-Level description of the development

The software prototype for this assignment has been implemented in the Julia programming language as a third party package called NMR.jl [1]. The package is hosted on github and can be installed painlessly from within julia - this process is described in the “Reproducing the results” section of this document.

There are two main entry points in NMR, **runjob** and **runcombiner**.

**NMR.runjob()** takes a NMR.Nmr type for its arguments which has a *jobid*, *inputs, fun, outputfilename*. This allows one to specify input files and a mapper/reducer function that the files are processed with, and then the result is stored as *outputfilename*. **NMR.runcombiner()** is then used to combine the results from any mappers/reducers into a single output file. Examples of this will be provided and elaborated on in the “Detailed description of the implemented MapReduce functions” section.

NMR automatically parallelises its work using separate *workers* based on the number of CPU cores it has access to. This is achieved by the **addprocs(Sys.CPU\_CORES)** call in test/runtests.jl which is executed before initiating any of the jobs. This being said, NMR can work with a single worker or an arbitrary number of workers, independent of whether it has access to a multi-core CPU or not.

Since the input files contain all the data for a job, they are split into N number of chunks depending on the workers in use. This is only done once, and then the mappers and reducers ran by workers operate on their respective chunks of data. For example the file test/data/AComp\_Passenger\_data.csv has 510 lines and when running the tests on a server with 4 CPU cores, this file is split into 4 files that reside in test/split/{2,3,4,5}/AComp\_Passenger\_data.csv (128, 128, 127 and 127 lines each respectively).

Below is an example diagram of a job that runs with 3 workers and has a mapper, reducer and a combiner. The split phase is only done once for any particular input file.

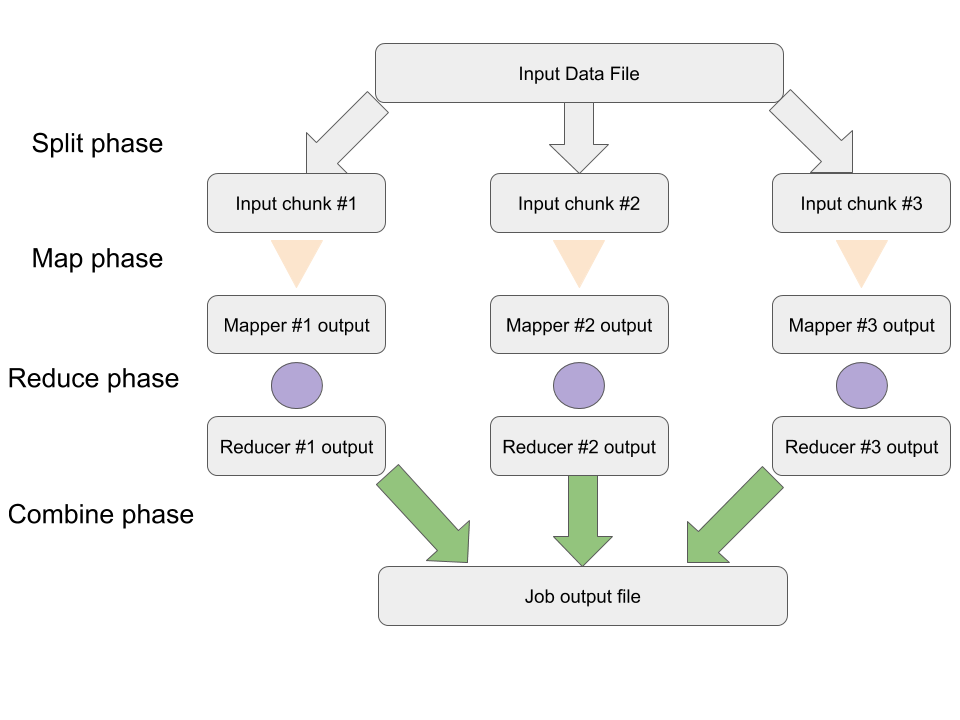


Figure 1: Example workflow using 3 workers

In the diagram above, the Split and Combine phases are performed by the master worker (coordinator), and the Map and Reduce phases are carried out by 3 separate workers in a parallel and asynchronous fashion.

# Reproducing the results

In order to reproduce the results, one needs to install Julia and the NMR.jl package that was developed for the purposes of this assignment. The code has been tested on MacOS and Linux but it is expected to work on Windows as well.

## Installing Julia

One can download Julia from its website [2]. Please download the 0.6.2 version of the language. There are binary packages for different operating systems.

## Installing the NMR package

Once Julia is installed, one should install the Naive MapReduce (NMR) package that was developed for the purposes of this assignment - using the Pkg.clone command as shown below.

➜ ~ julia -q

julia> **Pkg.clone("https://github.com/deyandyankov/NMR.jl")**

INFO: Cloning NMR from https://github.com/deyandyankov/NMR.jl

INFO: Computing changes...

INFO: Package database updated

julia>

## Running the objectives

Running the reports that have been implemented as part of this assignment is as simple as running the tests for the NMR package using the Pkg.test(“NMR”) command:

➜ ~ julia -q

julia> **Pkg.test(“NMR”)**

INFO: Testing NMR

INFO: Running job: NMR.Nmr(1, String["AComp\_Passenger\_data.csv"], NMR.mapper\_parserecordacomp, "acomp.csv")

[...]

INFO: NMR tests passed

julia>

A lot of the output has been omitted here. Those are INFO and WARNING messages related to error handling. The error detection and correction routines are discussed in the “Input data error detection and correction” part of this document.

The command above creates the output directory and the output files for the reports. On Linux and UNIX-like operating systems, this is normally **~/.julia/v0.6/NMR/test/output** - then, for each of the four objectives, there is a separate directory and output file. For example the output for Objective 1 is stored in **~/.julia/v0.6/NMR/test/output/1/output.csv**

# Simple description of the Version Control command line process undertaken

The Version Control process undertaken for the development of this project has made use of github [3] - a platform that allows free creation of git repositories for open source projects.

The location of the NMR repository is <https://github.com/deyandyankov/NMR.jl>

For the development, the simplest workflow has been used. I have initiated the repository locally and have committed and pushed to the **master** branch since there was no collaboration with additional developers involved and thus no merge conflicts were expected to occur.

A list of the different commits and stages of the development can be viewed on <https://github.com/deyandyankov/NMR.jl/commits/master>

# Detailed description of the implemented MapReduce functions

The implemented mappers, reducers, joiners and combiners work together to fulfill the objectives of the assignment.

For each objective, there is a test file that runs the necessary jobs in order to come up with its output.

In order to facilitate the review process, the outputs, in addition to all the code, has been committed in the repository and can be readily examined without the need of reproducing the results by installing julia and the NMR package.

## Objective #1

|  |  |
| --- | --- |
| Task | Determine the number of flights from each airport; include a list of any airports not used. |
| Inputs | test/data/AComp\_Passenger\_data.csv |
| Source code | test/test\_numberofflights.jl |
| Output file | test/output/1/output.csv |
| Header | airportid,flights |
| Example record | AMS,3 |
| Total records | 89 |

At the first execution of the job, the input file is split into a number of chunks based on the CPU cores. Then, the user defined mapper function NMR.mapper\_parserecordacomp is used to process it which stores its output in acomp.csv. This mapper parses the raw data into JSON records which are then easier to process. The user defined reducer NMR.reducer\_numberofflights is then applied on the acomp.csv output files from the mapper and the results from the reducers are stored in acomp\_flights\_reduced.csv (4 separate files assuming 4 CPU cores). Finally, those acomp\_flights\_reduced.csv files are combined by the user defined combiner NMR.combiner\_parsejson and the output is written to disk.

It is worth mentioning here that the NMR.mapper\_parserecordacomp skips a number of records that are deemed invalid. This is further elaborated on in the “Input data error detection and correction” section.

## Objective #2

|  |  |
| --- | --- |
| Task | Create a list of flights based on the Flight id, this output should include the passenger Id, relevant IATA/FAA codes, the departure time, the arrival time (times to be converted to HH:MM:SS format), and the flight times. |
| Inputs | test/data/AComp\_Passenger\_data.csv |
| Source code | test/test\_listofflights.jl |
| Output file | test/output/2/output.csv |
| Header | None, as the format is one JSON record per line |
| Example record | {"passengerid":"UES9151GS5","flightid":"SQU6245R","originairport":"DEN","dstairport":"FRA","departuretime":"2015-01-06T17:14:20","arrivaltime":"2015-01-07T10:43:20","totalflighttime":1049} |
| Total records | 479 |

This task is considerably simpler than Objective #1 since it does not involve a reducer. The same mapper NMR.mapper\_parserecordacomp is used to process the raw data and then the same combiner NMR.combiner\_parsejson is used to assemble the results and store output to disk. The reason why we do not need a reducer is that in the task description asks for the inclusion of “passenger Id”. This allows us to output one record per flight per passenger which translates to one or more output records per flight.

## Objective #3

|  |  |
| --- | --- |
| Task | Calculate the number of passengers on each flight. |
| Inputs | test/data/AComp\_Passenger\_data.csv |
| Source code | test/test\_numpassengers.jl |
| Output file | test/output/3/output.csv |
| Header | flightid,passengers |
| Example record | JVY9791G,5 |
| Total records | 122 |

Like Objective #1 and #2, Objective #3 uses the same mapper and combiner functions. A specialised reducer function, NMR.reducer\_numberofpassengers, is at work between the map and combine stages. This reducer function takes the output from the mappers, which is one JSON record per line, and only extracts the flightid and passengerid fields. Each reducer worker then spits out a hash with which has a flightid for its key and number of passengers for its value. The dictionaries output from the reducers are later merged in the combiner so that if data for a particular flight has been spread across the different mappers/reducers, the final number of passengers is the sum of the passengers detected by each reducer for every particular flight.

## Objective #4

|  |  |
| --- | --- |
| Task | Calculate the line-of-sight (nautical) miles for each flight and the total travelled by each passenger and thus output the passenger having earned the highest air miles. |
| Inputs | test/data/AComp\_Passenger\_data.csv test/data/Top30\_airports\_LatLong.csv |
| Source code | test/test\_numberofflights.jl |
| Output file | test/output/4/output.csv |
| Header | passenger,miles |
| Example record | UMH6360YP0,5671.48 |
| Total records | 33 |

Objective #4 required the most amount of work compared to the rest of the objectives, as it involved running mappers and reducers on two different input files and joining the result. The objective is achieved by multiple stages including mapping AComp\_Passenger\_data.csv, reducing it to a [flightid, [srcairport, dstairport]] format using NMR.reducer\_lineofsightpassenger and then combining it and storing it in lineofsightpassenger.json so that it can later be joined with the mapped output of the mapper NMR.mapper\_airportlatlon after it has processed Top30\_airports\_LatLong.csv.

The join operation was performed in userspace (see test file for details) and the Haversine distance has been used for the calculation of line-of-sight between the latitude/longitude coordinates of the origin and destination airports. The haversine() function is defined in src/haversine.jl and it uses 6.3781e6 meters as the Earth radius value which was derived from a wikipedia article [4].

The output of this objective is comprised of one line per passenger that includes their id and total miles travelled.

# Input data error detection and correction

The approach taken to facilitate error detection and correction was to use Julia user defined types for the necessary fields contained in the AComp\_Passenger\_data.csv and Top30\_airports\_LatLong.csv files. The code for error detection and correction is in src/udf/types.jl. We define a Passenger record as UDFRAcomp as a structure containing several fields: passengerid, flightid, originairport, dstairport, departuretime, arrivaltime, totalflighttime. Before instantiating an object of type UDFRAcomp, its components are built using their own types accordingly. Instantiating an object of a particular type in this case means we have control over validating and, if necessary, correcting the original value.

The only correction mechanisms in use are typecasting the value from string to integer when necessary (the case with total flight time) and also uppercasing all of the string values before validating them. One could potentially take further corrective measures like using string distances to fix typos in airport or identifiers but this was not part of the current implementation.

The following validations take place for the separate field types:

|  |  |
| --- | --- |
| UDFAirportName | Must be longer than 3 characters and shorter than 20 characters |
| UDFAirportCode | Must match ^[A-Z]{3}$ |
| UDFFlightId | Must match ^[A-Z]{3}[0-9]{4}[A-Z]$ |
| UDFPassengerId | Must not be an empty string  Must match ^[A-Z]{3}[0-9]{4}[A-Z]{2}[0-9]$ |
| UDFDepartureTime | Must match ^[0-9]+ |
| UDFTotalFlightTime | Must match ^[0-9]+ Must be less than 30 hours |

When any of the above conditions is not met, a WARNING is issued during job runtime and the passenger record is skipped. Several examples of this are included here:

|  |
| --- |
| WARNING: [JOB 1] mapper encountered an exception of type NMR.UDFException on line 97: Airport id must be comprised of three capital letters. Trying to initialise with SI| instead. |
| WARNING: Total flight time 1843 minutes is more than 30 hours |
| WARNING: [JOB 1] mapper encountered an exception of type NMR.UDFException on line 82: Invalid passenger id: |
| WARNING: [JOB 4] mapper encountered an exception of type NMR.UDFException on line 12: Invalid flight id: 4QU6245R |

Currently these messages are printed to standard output during job execution. In a future release they can potentially be stored as error logs and related to a particular job run.

When parsing AComp\_Passenger\_data.csv using the rules outlined above, a total of **117** errors were detected. This figure does not include the number of errors which were corrected after detection.

# Lessons learned

This coursework has provided an excellent opportunity to dive into the inner workings of the well known MapReduce paradigm for parallel processing. I have come to greatly appreciate the module leader’s structured approach to input data error detection and correction - it takes some determination and effort to set it up initially but in the end this way of data handling pays off massively. Additional learning outcomes include improved knowledge of julia’s package development and testing workflow as well as improved understanding of its toolset for handling parallel tasks which is part of the base language.

# Self appraisal

The developed solution is robust and well tested. The error handling mechanisms are clearly defined and easy to extend. The framework can potentially process very large amounts of data on a single server given adequate resources like multi-core CPU and sufficiently fast storage devices. Although there is some documentation in place, additional tutorials and examples must be put in place if the code is to be shared with a wider audience. Probably the best aspect of the framework is its loose coupling - the architectural components are strictly separated from the business logic. This design is meant to facilitate the development of a future version that uses networking to operate within not just one, but a number of servers in a cluster.

# References

[1] Naive MapReduce implementation, <https://github.com/deyandyankov/NMR.jl>

[2] Julia Downloads, <https://julialang.org/downloads/>

[3] github, <http://github.com>

[4] Earth Radius, <https://en.wikipedia.org/wiki/Earth_radius>