Advanced computing report

gv009864

Introduction

MapReduce, a programming model that is part of Apache Hadoop, allows big data to be processed via a parallel, distributed algorithm via clusters. The focus on this assignment was to create a prototype that focuses on the basic building blocks of the MapReduce. With a provided set of big data, in the form of csv files, a basic set of requirements needed to be met have been provided in order to illustrate an understanding of how a MapReduce system works.

Technologies used

Java was chosen as programming language of choice, mostly out of familiarity and experience that the author has with the language.

Gradle was chosen as the build automation tool, all dependencies that have been included in the project can be seen under the build.gradle file.

External libraries used were limited, but proved to be very useful. The libraries used for the assignment included:

Group	name	version
org.slf4j	sl4j-simple	1.6.1
commons-collections	commons-io	1.3.2
org.apache.commons	commons-collections	4.0

- sl4j logger to log events into the command-line during runtime
- commons-io to help deal with some of the streams being opened to read the csv data.
- commons-collections to help make the splitting of data much more trivial.

High level description of prototype

Hadoop's MapReduce uses three main stages in order to process data: mapping, shuffling and reducing. These stages are separate in order to help achieve the scalability that is required by those that want to process these vast quantities of data. Churning through that data quickly would also require an implementation of parallelisation that can run on a number of nodes where the nodes are connected to some form of network.

The prototype being developed in this assignment aims to focus on creating the Mapper, Shuffler and Reducer components that are able to process the csv data provided. Where Hadoop aims to run on clusters, this prototype is designed to just work on the system it is being run on as there is no requirement to include this feature.

The distinct stages implemented are:

- Parse and validate the passenger and airport csv data. This step will also need to remove any duplicate values that have been included in the input data.
- Split the data into chunks of n size
- Create n number of mappers to carry out a map job that creates a HashMap of KeyValue pairs.
- Shuffle stage to merge the data returned by all of the completed map jobs and output a single HashMap for passenger and airport data to feed into the Reducer.
- Reducer will consume HashMaps for passenger and airport data, then evaluate results from that data.

Version Control process

Git was used as the version control system for this project. The university GitLab was used to host the repository for this project.

Link to the CS-GitLab repository: https://csgitlab.reading.ac.uk/gv009864/advanced-computing-coursework

Commands used to commit and push changes to the repository

1. After setting up the Gradle project, `git init` was used in root directory to initialise it as a git project.
2. Set up the project to use the newly created repository on CS-GitLab:

```
git remote add origin https://csgitlab.reading.ac.uk/gv009864/project.git
```

3. Check the status of the project to see what state files in the project are in:
    - `git status`
4. Add files to staging with and commit those changes locally to save them:
    - `git add *` or `git add _filename_`
    - `git commit -m "First commit"`
5. Push any local commits and provide credentials as https is being used:
    - `git push`

Detailed description of the MapReduce functions being replicated

Mapper

Two Mapper implementations were required to handle mapping KeyValue pairs for both the passenger and airport data. This was a result of the solution required to handle parsing and storage either data type.

To increase performance, the mappers were created as threads to allow for parallelisation. This way, each data type can be split up into chunks of *n* size then be passed into a Mapper and have the run method be called.

```
PassengerData passenger = new PassengerData(CSV.PASSENGER_DATA.getClasspath());
AirportData airport = new AirportData(CSV.TOP_30.getClasspath());
LinkedList<PassengerMapper> mappers = new LinkedList<>();
LinkedList<AirportMapper> airportMappers = new LinkedList<>();

...

// Split data into chunks of the specified size
List<List<String[]>> passengerData = ListUtils.partition(passenger.getLines(), 20);
List<List<String[]>> airportData = ListUtils.partition(airport.getLines(), 5);
// Create n number of mappers depending on number of partitions made.
for(List<String[]> l : passengerData){
    mappers.add(new PassengerMapper(l));
    mappers.getLast().run();
}
// Do the same for airport data
for(List<String[]> l : airportData){
    airportMappers.add(new AirportMapper(l));
    airportMappers.getLast().run();
}
```

In the PassengerMapper, the following approach was taken to create the KeyValue pairs. In the for each loop, *s* represents a line parsed from the passenger csv data file. The data is expected to be of 6 elements, in size. If for some reason that the validation did not catch this, a check here ensures that the line is not processed by skipping to the next iteration.

A check is then done to see an element in the HashMap already exists with the desired key (the flightID). If the element is found, then a new passenger can be added under the flight object which is assigned as the value. If nothing is associated with the desired key (i.e. null) then a new HashMap entry can be created from the currently selected line *s*.

```
@Override
public void run() {
    super.run();
    // Want to iterate through all elements, easier to just do a foreach
    for(String[] s : this.data) {
        // Skip the line if somehow the data is not of the correct size.
        if(s.length != 6) {
            log.error("Loaded data entry is of incorrect size. " +
                "Expected {}, found {}.", 6, s.length());
            continue;
        }
        // First check if a flight KeyValue pair already exists.
        // If it does, just add a passenger to it.
        if(this.hashMap.get(s[1]) != null){
            this.hashMap.get(s[1]).addPassenger(s[0]);
        }
        // Otherwise add a new KeyValue pair to the HashMap as <FlightId, Flight>.
        else {
            this.hashMap.put(s[1], new Flight(s[1], s[2], s[3], s[4], s[5]));
            this.hashMap.get(s[1]).addPassenger(s[0]);
        }
    }
}
```

A simpler approach was taken for creating a HashMap from the airport data. Instead of checking for any existing entries, it will simply skip the entry. No duplicates should be present in the airport data.

```
@Override
public void run() {
    super.run();
    for(String[] s : this.airportData) {
        // Skip the line if somehow the data is not of the correct size.
        if(s.length != 4) {
            log.error("Loaded data entry is of incorrect size. " +
                "Expected {}, found {}.", 4, s.length);
                return;
            }
            if(this.hashMap.get(s[1]) == null){
                this.hashMap.put(s[1], new Airport(s[0], s[1], s[2], s[3]));
            }
        }
    }
```

Shuffle

Once all of the Mappers have been completed, all of the results need to be merged together into something that can be processed by the Reducer. For the mapped airport data, this is not an issue as the mappings are very simple nothing needs to be merged. This is not the case for the passenger data.

A decision was taken to store each passenger ID under the corresponding Flight object. This was due to logic of how a flight has a some details, but also hold multiple passengers. This is evident when looking at the raw csv data; lots of passengers share the same flight information. In order to merge the HashMaps produced by all of the mappers, the passenger lists need to be merged.

Java 8 introduced the *merge* method for HashMaps and was very good option to solve this issue. The following code basically allows for a merge method to be implemented where two values, with the same **key** are found. In this case, v1 and v2 will have the same key and the *mergePassengers* method can be called (part of the Flight class) that will merge the passenger lists together and return a new Flight object to be used as the value against that key.

```
newHashMap = new HashMap<>();
airportHashMap = new HashMap<>();

//Merge HashMaps
for(PassengerMapper m : mappers) {
    m.getHashMap().entrySet()
            .forEach(entry -> newHashMap.merge(
                    entry.getKey(),
                    entry.getValue(),
                    (v1, v2) -> v1.mergePassengers(v2.getPassengers())
            ));
}

//Merge airport HashMaps
for(AirportMapper m : airportMappers) {
    airportHashMap.putAll(m.getHashMap());
}
```

Reducer

The Reducer class will take in the the new passenger and airport HashMaps created after the shuffle step and the relevant methods have been put in place to produce the desired outputs. Each of public methods in the Reduce class return strings that can be printed out to a desired output stream; this prototype just prints to STDOUT.

Get source airports used

This method mainly focused on filtering the data via the use of a stream. A method prior would have been run, using both the passenger and airport HashMaps to determine the number of times an airport was used as a source location for a flight. The value of which is stored under each airport object stored in the HashMap. The values filtered were then stored in a StringBuilder and the contents returned once finished.

```
Stream<Map.Entry<String,Airport>> stream =
                this.airportHashMap.entrySet()
                .stream();

        if(flag) {
            log.info("Filtering for airports used once or more as source location for flight.");
            stream = stream.filter(x -> x.getValue().getNumOfFlightsFrom() > 0);
        }
        else {
            log.info("Filtering for airports never used as source location for flight.");
            stream = stream.filter(x -> x.getValue().getNumOfFlightsFrom() == 0);
        }
    ...
```

Get the list of flights

This method was trivial to implement as it just required iterating through each of the values in the passenger HashMap returning the value of the *toString()* method. Values stored in a StringBuffer and the contents are returned once finished.

```
    ...
    this.passengerMap.forEach(
            (key, value) -> builder.append(value.toString() + System.lineSeparator())
    );

    return builder.toString();
```

Get the number of passengers for each flight

An aim for this method was to return an ordered list as would be a much easier way to read and understand the data. Though just returning the number of passengers is trivial, an approach to treating the data as a stream to utilise the *sorted()* method was used. The method orders based on the integer value returned by *getPassengers().size()* and reverses the order to produce a list in descending order.

```
...
    this.passengerMap.entrySet()
            .stream()
            .sorted(Comparator.comparing(
                    a-> a.getValue().getPassengers().size(),
                    Comparator.reverseOrder()))
            .forEach( a -> builder.append(
                    a.getKey() + " : " + a.getValue().getPassengers().size() + System.lineSeparator())
    );
    return builder.toString();
```

Getting nautical miles covered by flight and passengers

In order to get the nautical miles covered by each flight (stored in the passenger HashMap), calculations need to be run against each latitude/longitude pair. Geodatasource provides a snippet of code for various language (including Java) that can carry out the calculations. For the sake of this prototype it will do as it can calculate the nautical miles (as require in the specification) that are desired. A flag can be passed into the method to return a value in that unit.

```
    // Distance method from https://www.geodatasource.com/developers/java
    this.passengerMap.forEach(
            (key, value) -> {
                Airport a = this.airportHashMap.get(value.getSrcAirportCode());
                Airport b = this.airportHashMap.get(value.getDestAirportCode());
                this.flightDistances.put(key, distance(a.getLatitude(), a.getLongitude(),
                                b.getLatitude(), b.getLongitude(), "N")
                );
            }
    );
```

Values are stored into a new HashMap that uses the flight code as the key and the distance as the value. The method that returns the value, sorts the values before returning the results in descending order to make it easier for the end user to read.

Error handling strategy

It was clear, from viewing the *AComp_Passenger_data.csv* file that that it was filled with many errors and duplicates. The strategy in place lies mostly in the data parser where the code is read and validated at the same time. The prototype has three classes under the *data* package that deal with loading the csv data.

The CsvData base class has a *loadFile()* implementation that is shared between the PassengerData and AirportData classes (which extend the CsvData class). The method is designed to validate each line as they are read. The implemented validator will confirm the validity of the read line, returning a boolean value to signify this. By default, in the CsvData class, the validator will simply return true (i.e. no implementation). PassengerData and AirportData each have their own validators that override the base class implementation which provides that data specific validator.

```
try {
    log.info("Reading data from {}", resourcePath);
    while((readLine = reader.readLine()) != null) {
        splitLine = readLine.split(DELIMITER);

        if (this.validateData(splitLine))
            data.add(splitLine);
        else
            log.error("Skipping {}", readLine);
    }
}
```

The following code shows the validator implementation in the PassengerData class. Regex pattern matchers are used to validate each of the values that will be present in the array. The validation adheres to what is stated in the specification. The Validator enum class is used to store all of the regex patterns is fed into a method in the base class that will iterate through each of the elements in the array.

```
@Override
protected boolean validateData(String [] data) {
    // Immediately fail validation if the array does not have 6 elements.
   if(data.length != DATA_LENGTH) {
        log.error("CSV only has {} element(s), expected {}", data.length, DATA_LENGTH);
        return false;
    }
    Pattern[] patterns = {
            Validator.PASSENGER_ID.getPattern(),
            Validator.FLIGHT_ID.getPattern(),
            Validator.AIRPORT_CODE.getPattern(),
            Validator.AIRPORT_CODE.getPattern(),
            Validator.EPOCH_TIME.getPattern(),
            Validator.FLIGHT_TIME.getPattern()
    };

    return this.matcherIterator(DATA_LENGTH, data, patterns);
}
```

There is also another block of code in *CsvData.loadFile()*, just after reading, that deals with removing any duplicate values that may have been loaded. Using the stream method to go through all of the data and utilising the merge method to return only a single instance of any two values that have identical passengerId values (hence the use of arr[0]).

```
this.lines = new LinkedList<>(
                data.stream().collect(Collectors.toMap(arr -> arr[0], Function.identity(),
                (a, b) -> a)).values());
```

The output format of any reports that each job produces

A full output from the execution of the application under `Output.md` can be found in root directory of the Git repository.

Number of flights from each airport

Full list of the number of flights from each airport with the airport code and associated name.

```
ORD : CHICAGO : 2
PVG : SHANGHAI : 1
CDG : PARIS : 1
CLT : CHARLOTTE : 1
JFK : NEW YORK : 1
DFW : DALLAS/FORT WORTH : 1
LHR : LONDON : 1
MUC : MUNICH : 1
BKK : BANGKOK : 1
KUL : KUALA LUMPUR : 2
MIA : MIAMI : 1
CGK : JAKARTA : 2
AMS : AMSTERDAM : 1
DEN : DENVER : 4
CAN : GUANGZHOU : 2
IAH : HOUSTON : 2
MAD : MADRID : 1
FCO : ROME : 1
PEK : BEIJING : 1
ATL : ATLANTA : 2
HND : TOKYO : 1
LAS : LAS VEGAS : 1

LAX : LOS ANGELES : 0
SFO : SAN FRANCISCO : 0
PHX : PHOENIX : 0
HKG : HONG KONG : 0
IST : ISTANBUL : 0
DXB : DUBAI : 0
FRA : FRANKFURT : 0
SIN : SINGAPORE : 0
```

List of flights

```
{id='QHU1140O', srcAirportCode='CDG', destAirportCode='LAS', departTime=17:14:58, totalFlightTime=1133}
{id='FYL5866L', srcAirportCode='ATL', destAirportCode='HKG', departTime=17:25:40, totalFlightTime=1751}
{id='BER7172M', srcAirportCode='KUL', destAirportCode='LAS', departTime=17:26:07, totalFlightTime=1848}
```

```
id='RPG3351U', srcAirportCode='HND', destAirportCode='CAN', departTime=16:59:29, totalFlightTime=374}
{id='KJR6646J', srcAirportCode='IAH', destAirportCode='BKK', departTime=17:26:43, totalFlightTime=1928}
{id='VYW5940P', srcAirportCode='LAS', destAirportCode='SIN', departTime=17:26:43, totalFlightTime=1843}
...
```

Number of passengers on each flight

Number of passengers on each flight in descending order.

```
XXQ4064B : 19
ULZ8130D : 18
WSK1289Z : 17
...
XIL3623J : 9
WPW9201U : 8
HUR0974O : 6
EWH6301Y : 6
PNE8178S : 1
```

Nautical miles covered by passengers

List of each flight, with the passengers, with the nautical miles covered. Output is listed in descending order, with the flight and passengers that covered the largest distance at the top.

```
FlightId: YZO4444S
Nautical Miles: 8430.447260331151
Passengers: 15
WBE6936NU4
CYJ0225CH9
...

FlightId: KJR6646J
Nautical Miles: 8019.5054259044555
Passengers: 15
WBE6936NU3
JBE2303VO1
...

FlightId: BER7172M
Nautical Miles: 7688.481122976476
Passengers: 11
WBE6936NU2
BWI0521BG0
...
```

Self-appraisal

I have quite enjoyed doing this assignment as it has allowed me to create some interesting code. Implementing a non-MapReduce prototype has given me a better understanding of how Hadoop's MapReduce function underlying code actually works. Though, not a perfect solution by any means, this prototype still implements the three distinct steps that MapReduce provides.

After writing this report, I feel that all the validation code should be moved from the parsing stage and into the mapping stage. Reason being is that this can help vastly increase performance especially when you consider the affect larger data sets can have to performance.

It is hard to gauge how well the code work with much bigger data how well it could potentially scale without the data. I feel that some steps, such as the reduce functions that rely on using streams, don't take advantage of extra parallelisation. Though I cannot imagine it being too difficult to modify in order to take advantage of this if given the time to do so.

I think it would be really interesting to see if there is a way to generate a larger data set and see how well this implementation would work with a data set size that is more typical to see in industry.