BIG DATA PROGRAMMING CODING PROJECT

for

BASH, HADOOP, SPARK, GRAPHX

Version 1.0

Adam Ryan (14395076)

COMP47470

December 8, 2021

Contents

1 Introduction

1.1 Purpose

As part of Big Data Programming, assignment 2 requires students to complete four individual assignments covering programming tasks in the four key technologies discussed within the course to answer various questions, and complete a report on the solution. This document is designed to capture the answers to these questions, and detail the code used.

1.2 Note on Ordering

In answering some of these questions, I did not always answer the questions in the order of the questions, as in some cases it was more sensible to do later questions first (particularly in the Bash project). For this reason, the order of answers given will not always match the assignment order.

1.3 Note on Computer

This was ran on an M1 Mac and as such the docker files given in the labs for everything except the MySQL step don't without changes work so the environment setup used to complete the assignment is different to the labs. Please configure the scripts appropriately for your machine/setup.

2 Bash and Data Management - 30%

3 Pages used out of 10 pages. The answers are contained in the question one folder.

2.1 Cleansing

For this section, please refer to the script named q1_part1_cleaning.sh in the question 1 folder for my code.

2.1.1 Reading in the CSV with delimeter

The first part which is completed is the reading in of the CSV. In this part we need to account for the presence of triple-barrel names which match the overall file delimiter.

The initial cleanse function starts by taking a file and copying it as a new staging file given as arguments. It uses grep to find entries in the CSV which are wrapped in Quotes and contain the delimiter '-'. It then uses sed and the same regex to substitute instances of the - with a space; this is designed so that triple barreled names such as "Adam Joseph-Patrick-Ryan" is transformed into "Adam Joseph Patrick Ryan". This is then saved as a 'staging' file. This is done first so that any field which is wrapped in quotes containing a delimiter is removed at the beginning to ensure that I can then read in the file properly.

2.1.2 Removing the erroneous character from Name

The file which has been cleansed in the previous file is then read in using the function initial_formatting_q1 (the previous function described is contained in this) as a CSV. As we've dealt with in-field delimiters, we are safe to read in the fields appropriately according to their column position. Each line of the CSV is read with the delimiter, and each field is taken as a variable.

For the fix on the name field, to remove the erroneous character, whatever is read in is passed to a function defined name_fix which uses standard bash functions (instead of regex) to replace the first occurence of the erroneous entry in name with a blank string (so it's removed) and then it removes all instances of the surrounding quotation on the name field which have been 'left' from the previous section.

2.1.3 Country Name Fix

While the fields are being read in (as in the previous section) the country field is passed to the country map function. This takes the value from the CSV field and searches for the country code in the CSV file. It takes the country name field (using the file delimiter) and returns that as the country name to be used.

2.1.4 Two columns with non-useful values

With each line having been taken with the fixes on the relevant variable implemented, the full line is then echoed to a secondary cleansed staging CSV. We remove the first cleansed staging file as we no longer need this (but can choose to keep it for auditting if we wish by removing the rm call)

To determine which columns contain non-useful values in a systematic way, the function removing_single_value_fields_q1 is called on the output of the previous part.

The logic here is that we want to build up a string of the column indices which we want to keep that have more than two unique values in the field (including the header) to take only these columns from the csv.

For each column in the file it takes the number of unique values in that field. If the number of unique values is greater than two (the header is included in the count, hence value of one or two implies either empty but with a header or a single unique value with the header) we add that to the list of fields we want to keep comma separated. Otherwise (if the column has two or one unique values) we exclude it.

Finally, if there are fields to keep (the string of fields to keep is non blank) then we write that to our final CSV and remove the previous staging CSV (but again can choose to keep it if we want for auditting by removing the script).

We are now left with a cleansed CSV file fulfilling this question.

This entire process is wrapped in the main_run_q1 function.

2.2 SQL

The SQL script is contained in the file named q1_mysql.sh. It is assumed as a prerequisite that the previous part has been ran before running this.

2.2.1 Create the Tables

First to allow it to be re-run it drops any database and then creates one with my student number and uses this in subsequent parts. Similarly it drops the bash_dm table on this database if it exists and subsequently creates it using the create table function passing in the columns and types. The index column is taken as the PK.

Next it reads in the cleansed CSV from the end of the previous part and inserts each row into the table; the header is excluded from the insertion by checking if the first value is labelled as 'INDEX'. This same functionality could also be achieved by reading in the whole file and then doing a bulk insert, however I've chosen to do a line-by-line commit as the assignment doesn't outline one preference over another.

2.2.2 Average Height by Country.

It selects the average height grouped by country and outputs this to the avg_height.csv. Please refer here for the answer.

2.2.3 Max Height Hair.

It selects the max height grouped by hair colour and outputs this to the max_hair_-colour.csv. Please refer here for the answer.

2.3 MongoDB

The SQL script is contained in the file named q1_mongodb.sh. It is assumed as a prerequisite that the two previous parts have been ran before running this.

2.3.1 Creation

First it imports the CSV into MongoDB test instance and denoting it has a header line and names the document bashdm.

2.3.2 Adding Second PK

Next it takes gets all instances in the bashdm document and sets up a cursor to run over these entries. It sets the initial count as zero and while the cursor has more results it incremenets the count and updates the next item in the cursor to include a field named row_id which captures the count, and then the cursor increments. This results in all entries having an incrementing column. As INDEX is zero indexed, while this row_id starts at 1 for the first record, these entries are essentially index+1.

2.3.3 Find the name of the person with lowest height

I aggregate and take min on height while returning all other values in the record (essentially getting the 'row' with the minimum height). **The minimum height is 139 from Ryan Hall in Ethiopia**. The method written will return all instances with somebody with the minimum height.

3 Hadoop Graph - 20%

5 Pages used out of 10 pages. The answers are contained in the question two folder.

3.1 Number of Routes connecting each Harbour

Refer to Assignment2Q1.java and the output text file for the answer. This is equivalent to a word count of the Harbour. In the map function I emit the port and count (starting at one), and the reduce function simply adds up the number of times the port is found with the count. This is almost exactly identical to the standard word count but taking only the port.

3.2 Wolfsbane Nine route's Harbour

Refer to Assignment2Q2.java and the output text file for the answer. **It's associated with harbour Mintcream-Tau**.

This is again almost equivalent to the standard map function. The value is irrelevant and we only care about the key. If the map function gets a route which is equal to Wolfsbane Nine it emits the harbour and 1 as the value. The reducer sums the occurrences of the value (but obviously only one instance is found so it just takes 1).

3.3 Carnation Sixty Seven route's Harbour

Refer to Assignment2Q3.java and the output text file for the answer. **It's associated with harbour Lightcoral-Pi and Seashell-Nu**.

This is equivalent to the previous answer except the route name is different. In the mapper any instances of that route emit the harbour with 1.

3.4 Emergency Routes

Refer to Assignment2Q4.java and the output text file for the answer. The emergency routes are

1. Bisque-Mu
2. Burlywood-Epsilon

3. Darkkhaki-Zeta
4. Ghostwhite-Omicron
5. Goldenrod-Sigma
6. Mediumvioletred-Eta
7. Midnightblue-Rho
8. Sandybrown-Iota
9. Seashell-Zeta

In the mapper check if the route has more than three characters. If yes and they start with 911 emit the harbour, otherwise don't.

The value, again, is irrelevant for the list of harbours so the reducer is unchanged.

3.5 Midnightblue-Epsilon's Connected Harbours

Refer to Assignment5Q5.java and the output text file for the answer. The other harbours are

1. Teal-Beta
2. Orangered-Beta

This is a bit trickier to do in one step than the previous questions. First, I change the mapper and reducer so that the key and values are text strings.

The overall challenge here is that we need to get the route no of the harbour, and then get any harbours with the same route no. In SQL, this is easy to do with a join but slightly trickier here.

To do it, in the map phase I emit all route nos as the keys, and then the harbour as the value.

In the Reduce phase, I build up a pipe delimited string. For each value (i.e. the harbour) against that route number, I append it onto a 'result string' and add pipe as a separater. If the result string contains the Midnight Blue Epsilon Harbour, then the reducer writes that string.

The output is then the route no and all harbours associated with that route, pipe separated, where one of the harbours is midnight blue epsilon, i.e. all other harbours which it is connected to by route. The answer can then be taken as all the non midnightblue epsilon harbours in this output.

4 Spark - 20%

7 Pages used out of 10 pages. The answers are contained in the question three folder. I use SQL for most parts because it's easy, however alternatives using the DF are possible. Please refer to scala.txt for the code.

4.1 Records in Set

Read in the dataframe from the csv file noting there are headers. Use count to count the rows. Answer: 1000.

4.2 Restaurant with max reviews

I register a temptable named dfTable which allows me to query this using SQL. I cast No.Reviews as an int and then order this field and take the first result found (i.e. max count). This is the **Roasted Shallot** with 1500 reviews.

4.3 Longest name

I run an SQL query selecting the row ordered by the length of restaurant field in descending order and taking the first result to give the restaurant with the longest name which is **Extraordinary Vegetable Soup Emporium Place**.

4.4 Reviews per Region

I run an SQL question which sums the number of reviews (casted as an int) grouped by region.

4.5 Review Term Frequency

This is the only tricky question so I've included the code here.

The code (line separated by dots to fit here but one line in the answer file) is:

```
val wordCount_df=df
.withColumn("ReviewText", explode(split(trim(col("Reviewtext")), " ")))
.groupBy("ReviewText")
.count()
```

```
.sort($"count".desc)

wordCount_df.filter(!col("ReviewText").isin(Seq("The","and","of"):_*)).show
```

First, it creates a word count dataframe. Using the original dataframe, it takes the original Reviewtext Column, it removes trailing and leading spaces and then splits the column by string. The explode function then takes rows for the output of the split and it groups them into a new column called "ReviewText" (which now captures the terms in the reviews) and groups by this column and gets the count of occurrences (i.e. the term frequency) and sorts this in descending order by the count. The final line then filters this new word count dataframe and filters out the terms given and shows the result. As the terms given have difference capitalisation, I assume we should not lower case the reviewtext column as a preprocessing step. With this in mind, the top terms are

- the - 405 (note lowercase so not filtered)
- was - 295
- I - 295
- a - 228
- to - 216

Note that I interpretted the question as to get the term frequency and then filter out those words. Another reading of the question would be to filter out any of the review text with the terms and then get the term frequency on the remainder. If this was desired the it can be easily achieved by, before the first step, running an SQL query on the DF (and saving the result) where you take df rows where reviewtext not like %The%, not like %and%, and not like %of% and then run everything else as above (appropriately renaming the df in the val wordCount_df calls).

5 Spark GraphX - 30%

9 pages used out of 10 pages. Please refer to the question four folder and the graphX.txt file for the code. I use the Edge dataset (which is NOT equivalent to the other) which was ORIGINALLY uploaded.

5.1 Import the Data and Generate a Graph

Import the relevant modules. Define a Class called Trip for the CSV structure. Read in the Edge set CSV as a TextFile and remove the header row. Define a function which parses the CSV and makes each line a Trip item, and get an RDD of Trips.

Next, because an Edge needs to use Longs, but From and To are strings in our instance, I define a map that takes all the harbours, maps them to some autogenerated ID (we don't need to connect it back to the other dataset) and develop the inverse map so that I can take a harbour string and map it to a long. I do this on the list of harbours, and then I define the edges as the distinct sequence of from Harbour IDs to to Harbour IDs. I then define this as a graph.

5.2 An array of each harbours routes

I collect neighbours to generate for each harbourid the connected harbours. The routes are necessarily therefore that harbourid to the other harbourid.

5.3 Harbours for Route Porium Thirty-One

Porium Thirty-one is not present as a route; we see the inequivalence of the two datasets in this question as the other dataset has it connected to Yellowgreen-Eta and nothing else (eg. it's a self-connected vertex which doesn't really make sense in context). Checking the route is not present can be done by trivially filtering the RDD to this route name and seeing nothing is returned.

5.4 Harbour with most routes

I use the indegrees reduce function, as outlined in lectures, to show the harbour with the most routes, and then map the harbour ID back to harbour name using my inverse map. The value is Oldlace-Omicron.

5.5 Harbour with most other harbours

This is the exact same question as in 5.4. Obviously if a harbour is connected to another harbour by a route, there exists an edge connecting the two harbours. Therefore, the harbour with the most other harbours is the harbour with the most edges (as in the edge data set we only have edges when there's a route connecting two harbours; the question is equivalent to asking which harbour has the most edges/routes [as no edge exists in the edge set if there's not a route from one harbour to the next]) which is the same question we answered in the previous question as Oldlace-Omicron.

As I'm assuming there must be a mistake in the question rather than repeating the question twice, I've also provided a way to generate a list of all harbours in the largest connected subgraph (which consists of 59 vertices in total) by getting the connected-Components() to cluster the vertices, and then converting this into a DF aggregating to get the cluster with the largest number of connected components, and filtering the RDD to this value to show the list of harbours.