BASH AND DATA MANAGEMENT

Downloaded the **'bashdm.csv'** dataset using the provided link.
wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1FDqa2hJ9rPhLG1YEcvHlfUliLym1ul4u' -O bashdm.csv

Downloaded the dictionary for country code and country name.
wget --no-check-certificate 'https://docs.google.com/uc?export=download&id=1h62yHjiRXMElbusQdCOM6W0_kgr-sQiv' -O dictionary.csv

1.1 Clean-Up Tasks

1. **There is an error in the name column - every name starts with an erroneous string "#]". Write a command/script to remove this string from the name column.**
   sed -i 's/\#]//' bashdm.csv
   sed -i 's/\"//g' bashdm.csv

   The sed command is used to replace strings using a pattern specified inside single quotes. Instead of printing to standard output, the **'-i'** option allows you to modify the **bashdm.csv file.**

2. **Currently the database is delineated by a"-" character. Write a command/script to convert this file into a comma delineated file.**

   sed -i 's/James Patch\-Walker\-Willis/James Patch Walker Willis/' bashdm.csv
   sed -i 's/Veronica Rhodes\-Coppern\-Braiz/Veronica Rhodes Coppern Braiz/' bashdm.csv
   sed -i 's/Ann Brookshire\-Willsworth\-Swain/Ann Brookshire Willsworth Swain/' bashdm.csv
   sed -i 's/Tracy Briggs\-Johnson\-Carver/Tracy Briggs Johnson Carver/' bashdm.csv

   This is a necessary step before dealing with difficulties such as the format of the name separated by '-' The sed -i command was used to remove the '-'.
   sed -i 's/\-/\,/g' bashdm.csv

   Using the sed -i command, "-" is replaced with a comma in the above code. The backslash is used to get around the '-'.

3. **There are two columns in this dataset with non-useful values. Write a script/command that identifies the columns that do not change, and removes them from the output.**

   cut -d”,” -f1-6 bashdm.csv > bash.csv
   cp bash.csv bashdm.csv
   The cut command splits the data using a comma as a delimiter and moves [1-6] columns to a new file called 'bash.csv.' The "cp" command copies bash.csv's contents to bashdm.csv.

4. **Lastly, the country names in this dataset have been incorrectly added as country codes. Using the dictionary file provided (Link above) find and replace each country code with its correct country name**

```
while read -r line
do
        Code=$(echo $line | cut -d, -f 4)
        Country=$(grep $Code dictionary.csv | -d '&' -f 3)
        sed -i "s|${Code}|${Country}|" bashdm.csv
done < "bashdm.csv"
```

   We use the read command to parse the "bashdm.csv" file and the cut command to store the country "Code" for each line. Then, using grep, we look for the corresponding nation name in the "dictionary.csv" file and save it in the "Country" variable. Then, in bashdm.csv, use the sed tool to find the line containing the code and replace it with "Country." Carry out these steps for each line in the csv file.

1.2 Data management Tasks

1. **Create 2 Bash scripts, one for SQL and one for MongoDB, that inserts your cleaned records into the respective database software. For SQL, you should generate the table within your Bash script.**

   The database 'Countrydb' and table 'bashdmTable' are created using "mysql". Parse and split each line in the "bashdm.csv" file with cut, then insert into the table using insert query.

   Switch to the mongo shell and type "use countryMongo_db;" to create a new database in MondoDB. "db.createcollection("pawansTable")" follows. Then run the script countryMongoDB.sh, which inserts rows into tables using "mongo database_name –eval insert," which is equivalent to SQL insertion.

2. **Find the average height per country.**

```
mysql -D Countrydb -e "select Country, avg(Height) from bashdmTable Group By Country;"
```

For finding the average height per country, I grouped the table based on country using 'GROUP BY Country' and displayed the country vs avg(Height) per country.

Country	avg(Height)
Malaysia	161.2500
Qatar	165.1667
Ireland	172.8571
Kiribati	167.2500
Colombia	165.6667
Uzbekistan	168.3333
Poland	171.3333

The above are the top list of the country and it's average height.

3. **Find the maximum height per hair colour**

```
mysql -D Countrydb -e "select Hair_Colour, max(Height) from bashdmTable Group By Hair_Colour;"
```

For finding the Max height per hair colour, I used the same GROUP BY clause on Hair colour and used MAX operation of height to display max height per hair colour.

Output:

Hair_Colour	max(Height)
White	200
Black	200
Dyed	199
Ginger	199
Blonde	200
Brown	200

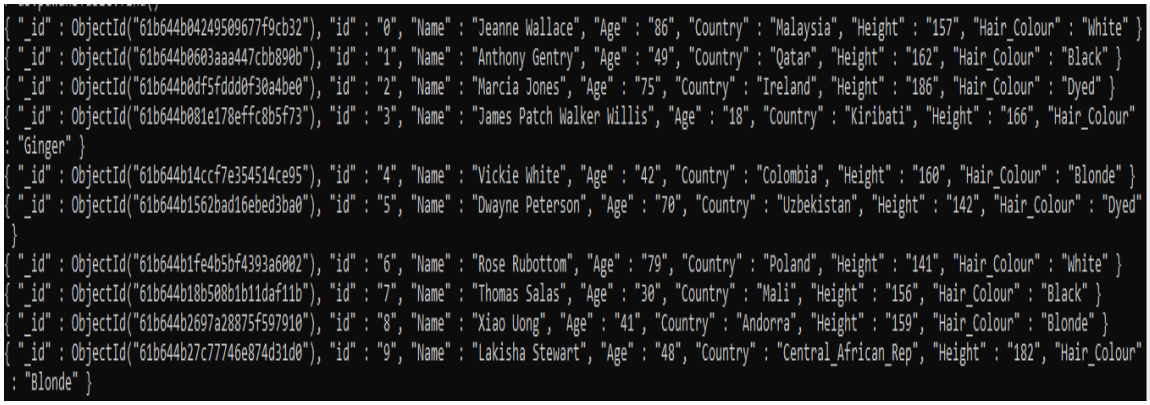
4. **(MongoDB) Write a short script that adds a new characteristic to each person - an ID number. You may generate this number however you like (e.g. a counter) within your script.**

For adding a new characteristic, I used the already existing column data in the dataset for ID.

idx=$(echo $line | cut -d ',' -f 1)

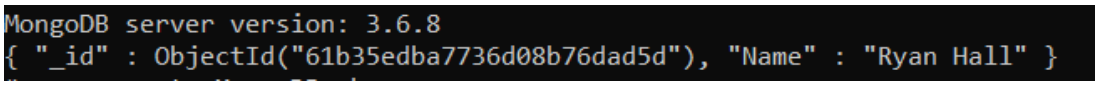
I performed cut operation to access the column and inserted it into the MongoDB as the separate column.

The below snap shows the "id" column added into MongoDB.



5. **(MongoDB) Find the name of the person with the lowest value for height.**
   use countryMongo_db
   db.pawansTable.find({},{Name:1}).sort({Height:1}).limit(1)

   Using the above query , I found the person with the lowest value for height. I sorted the rows based on height in ascending order and did limit of 1 to get the lowest record. When you run this query, you get the below output.



2. HADOOP

1. Create a list of number of routes that connect to each harbour

com.sun.tools.javac.Main RouteHarbour.java

The first command will act as a compiler for the "RouteHarbour.java" java program.

jar cf pf.jar RouteHarbour*.class

After constructing the java application, the second command will create a jar for the RouteHarbour class.

hadoop jar pf.jar RouteHarbour /input /output

The third command will define the generated jars' input and output folders, allowing the mapper and reducer classes to interact with them.

hdfs dfs -cat /output/part*

For all sections, the fourth command will execute it and place the output in the "output" directory. It will essentially show a list of the number of routes that connect to each harbour.

2. Create a list of the number of routes that connect to each harbour

hadoop com.sun.tools.javac.Main RouteHarbour.java
jar cf pf.jar RouteHarbour*.class

hadoop jar pf.jar RouteHarbour /input /output_rh
hdfs dfs -cat /output_rh/part* | grep "Wolfsbane_Nine"

The RouteHarbour java program is executed in the same way as before, but this time it checks for the Wolfbane Nine Route and emits that list.

Output – Mintcream-Tau

3. What harbours are connected by route Carnation Sixty-seven(No.1223).

hadoop com.sun.tools.javac.Main RouteHarbour.java
jar cf pf.jar RouteHarbour*.class
hadoop jar pf.jar RouteHarbour /input /output_rh
hdfs dfs -cat /output_rh/part* | grep "Carnation_Sixty-seven"

The RouteHarbour java program is executed in the same way as before, but this time it checks for the "Carnation Sixty-seven" Route and emits that list and produces **Lightcoral-Pi, Seashell-Nu as output**

4. Which harbours fielded emergency routes - these are routes whose route number begins with ”911”

hadoop com.sun.tools.javac.Main Find_nine11.java
The first command will act as a compiler for the " Find_nine11.java" java program.

jar cf noof.jar Find_nine11*.class
After constructing the java application, the second command will create a jar for the Find_nine11class.

hadoop jar pf.jar RouteHarbour /input /output911
The third command will define the generated jars' input and output folders, allowing the mapper and reducer classes to interact with them.

hdfs dfs -cat /output911/part*
For all sections, the fourth command will run it and place the output in the "output911" directory.

It will essentially display the following list of harbours that have emergency routes:

Midnightblue-Rho, Mediumvioletred-Eta, Goldenrod-Sigma, Darkkhaki-Zeta, Seashell-Zeta, Bisque-Mu, Burlywood-Epsilon, Sandybrown-Iota, Ghostwhite-Omicron, Midnightblue-Rho, Mediumvioletred-Eta, Goldenrod-Sigma, Darkkhaki-Zeta, Seashell-Ze

5. **”Midnightblue-Epsilon” is connected to two other harbours by a route- what are they? As a hint, consider how you could compare multiple MapReduce outputs with different Map conditions to find the link. For this question you do not have to combine your separate operations into one script.**

   hdfs dfs -cat /output/part* | grep "Midnightblue-Epsilon"

   output- Chrysanthemum_Four_hundred_and_sixty_five .

   hadoop com.sun.tools.javac.Main DifferentHarbours.java
   jar cf oh.jar DifferentHarbours*.class
   hadoop jar oh.jar DifferentHarbours /input /output1
   hdfs dfs -cat /output1/part*

   Output - Midnightblue-Epsilon, Orangered-Beta, Teal-Beta

   Here, I first examined whether routes are connected to Midnightblue—epsilon, and then used a different java application to find the true harbours for that route.

3. SPARK

Using the commands below, we'll launch two containers, one for the Spark master and one for the Spark worker.

The Spark master and Spark worker containers will be built when you perform these instructions. Now, inside the spark master, we need to use the following command to get the dataset that is required for this problem.

wget --no-check-certificate
'https://docs.google.com/uc?export=download&id=1Kay7TkrEr-3u1Q34ObOuEhhibpwAaMPj' -O spark.csv

Then, by performing the command below, we can load the spark shell.
/spark/bin/spark-shell

Read Data from file:

You will use a dataset of restaurants and reviews in this section. [INDEX, Name, Region, Number of Reviews, Review Text] is the format of the dataset.
val data = spark.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema","true").load("/spark.csv")

Spark includes a read operation with format and modifier options. We use the load function to give the source file'spark.csv' in a 'csv' format and arguments like 'header,true' to take the column names.

data.registerTempTable("table")

Using the 'registerTempTable' command from above, a table will be created.

1. **Determine the number of records the dataset has in total**.

```
scala> val counts = spark.sql("select count(*) from table")
counts: org.apache.spark.sql.DataFrame = [count(1): bigint]

scala> counts.collect.foreach(println)
[1000]
```

We can now use'spark.sql' to run any sql command on the 'noaatable' using the parentheses. The number of records in this table is provided by count(*), and the result of this query is saved in the sorted variable. The contents of the variable will be displayed by the following command collect.

2. **Find the restaurant with the highest number of reviews.**

```
scala> val noRev= spark.sql("select Restaurant,`No.Reviews` from table order by `No.Reviews` desc limit 1")
noRev: org.apache.spark.sql.DataFrame = [Restaurant: string, No.Reviews: int]

scala> noRev.collect.foreach(println)
[Roasted Shallot,1500]
```

I just altered the SQL query for this question. I used the limit property to take the first row from the 'No.Reviews' column, which was arranged descending. Then, using the select attribute, I created an array with the Restaurant and No.Reviews, which I saved in the RevNo variable.

3. **Determine the restaurant with the longest name.**

```
scala> val RestaurantName= spark.sql("select Restaurant from table order by length(Restaurant) Desc Limit 1")
RestaurantName: org.apache.spark.sql.DataFrame = [Restaurant: string]

scala> RestaurantName.collect.foreach(println)
[Extraordinary Vegetable Soup Emporium Place]
```

For this question, I used 'length(Restaurant)' to rank the 'Restaurant' column in the 'noaatable' in descending order, taking into account the number of characters inside the cell. Then I used the Limit Function to get the first element.

4. **Find the number of reviews for each region.**

```
scala> val region= spark.sql("select Region,sum(`No.Reviews`) from table group by Region")
region: org.apache.spark.sql.DataFrame = [Region: string, sum(No.Reviews): bigint]
```

I categorized the information by 'Region' and provided the Region sum for this question (No.Reviews). This generated a list of total reviews by region, which was saved in the region variable.

5. Determine the most frequently occurring term in the review column that doesn't include one of the following stop words: "A", "The", "of".

dataRDD.map(line => line.split(",")(4)).filter(line=> !(line contains("The")) && !(line contains("of")) && !(line contains("a"))).flatMap(line => line.split(" ")).map(word => (word,1)).reduceByKey(+).sortBy(T=> T._2, false).first()

Output: (String, Int) = (and, 279)

I parsed the'spark.csv' file and saved it in the noaaDataRDD variable for this inquiry. Then I made a 'filtered' variable that includes all words in the reviews except "The,of,a." By mapping word, count of each word to this variable, I was able to build a word count for each one. By adding the + operator to value, I was able to decrease the word string using'reduceByKey.' Our RDD yielded a list of unique words and their counts. Then I sorted it and showed the first one.

4. GraphX

For this section, I chose the below dataset with [port, portNo, Route, RouteNo]

```
wget --no-check-certificate
↪ 'https://docs.google.com/uc?export=download&id=19Uubzr_jcXGiVse5EJCOmBpLNEHH7YXY'
↪ -O hadoop_mirrored.csv
```

1. Import the data and create a graph representing the data

I made a graph with the vertices being harbours and the edges being routes between harbours. Port Name is a property on each vertex, using PortNo as the Id. Each Edge has a No for the source and destination harbours, with the destination being RouteNo. The destination Port Name, which is present in Route, is the edge's property.

I made a 'harbours' RDD with two fields: portNo and portName. I constructed the 'edges' RDD, which has the properties portNo, RouteNo, and Route. As a default vertex, I've named it 'nowhere.' I built the graph below using these three variables as parameters.

```
scala> val graph =Graph(harbours,edges,nowhere)
graph: org.apache.spark.graphx.Graph[String,String] = org.apache.spark.graphx.impl.GraphImpl@2f57b8
```

2. **Generate an array of each harbour's connected routes - consult the spark documentation to identify the most suitable method for this. Alternatively, you may use Spark commands to generate this information.**

```
scala> graph.triplets.foreach(println)
((27,Cattleya_Three_hundred),(9492,Khaki-Epsilon),Khaki-Epsilon)
((37,Cosmos_One_hundred_and_one),(7293,Beige-Alpha),Beige-Alpha)
((41,Rattlesnake_One_hundred_and_eighty-three),(7334,Green-Beta),Green-Beta)
((44,Gladiolus_Two_hundred_and_fifteen),(7106,Orangered-Chi),Orangered-Chi)
((45,Narcissus_Four_hundred_and_eighteen),(5207,Lightgray-Iota),Lightgray-Iota)
```

In GraphX, a triplet view is also accessible. The triplet view logically combines the vertex and edge attributes, resulting in an RDD[EdgeTriplet[VD, ED]] holding instances of the EdgeTriplet class. Using this with our graph, an array of each Harbour's related routes will be generated.

3. **Which harbour(s) is/are served by route "Porium_Thirty-one"?**

```
scala> val har = graph.edges.filter { case (Edge(harbourno,routeno,route)) => route=="Porium_Thirty-one" }.take(5)
har: Array[org.apache.spark.graphx.Edge[String]] = Array(Edge(8516,1255,Porium_Thirty-one))

scala> harbourMap(8516)
res7: String = Yellowgreen-Eta
```

The harbour that takes this path will be determined by graph edges. We acquire the edge details by filtering the edges property of the graph based on the "Porium Thirty-one" route. We can now use the harbourMap variable to determine the harbour name based on the harbour id.

Following the execution, we discover that "Porium Thirty-one" is the server for "Yellowgreen-Eta."

4. **Which harbour has the most routes associated with it?**

```
scala> val assRoutes = graph.degrees.collect.sortWith(_._2 > _._2).map(x => (harbourMap(x._1), x._2)).take(2)
assRoutes: Array[(String, Int)] = Array((Lily_Three_hundred_and_ninety-one,12), (Darksalmon-Zeta,12))
```

The degree of each vertex will be provided through graph degrees. This will result in the most routes being associated with it. To reach the desired result, this was then sorted and mapped with 'harbourMap'.

Most routes are related with "(Lily Three hundred and ninety-one,12), (Darksalmon-Zeta,12)."

5. Which harbour is connected to the most other harbours?

```
scala> graph.collectNeighborIds(EdgeDirection.Either).map { case (x,y) => (x,y.size) }.sortBy( _._
2,ascending=false).map( x=> (harbourMap(x._1),x._2)).take(10)
res14: Array[(String, Int)] = Array((Lily_Three_hundred_and_ninety-one,12), (Darksalmon-Zeta,12),
(Gerbera_Sixty-six,10), (Fuchsia-Nu,10), (Mediumvioletred-Upsilon,10), (Campanula_Two_hundred_and_
twelve,10), (Lisianthus_Four_hundred_and_ninety-nine,10), (Ginger_One_hundred_and_thirty-two,10),
(Phalaenopsis_Fifty-four,10), (Narcissus_Forty-five,8))
```

The property 'collectNeighborIds(EdgeDirection)' in Graph returns a list of all the vertices that are neighbors. 'EdgeDirection.Either' was used to examine both the indegree and outdegree. After that, I sorted it and mapped it to harbourMap to get "(harbourName, Count(NeighborIds)".