Project 2: Analyzing Data using Spark (Batch Processing)

Objectives

- Practice loading large data.
- Practice analyzing data using RDDs.
- Practice working with structured data in Spark.
- Running code on an AWS cluster.
- Working in groups and managing time.

Code

You will need to add your code to the provided project template available at: `https://github.itu.dk/omsh/BDM2019/tree/master/Projects/BIDMT_F19_P2`
Note: Make sure that you do not change the signature of any of the functions given to you. You can certainly add your helper functions, but we will test your code by calling the ones in the given code template.

Datasets

You will be using the two datasets, follow these steps to download them:

- **Airline Delay and Cancellation Data, 2009 - 2018**. The dataset is available from Kaggle. You can download it download it from this AWS S3 bucket to your AWS cluster.
- **Yelp**. The dataset is available from Kaggle. You can download it download it from this AWS S3 bucket to your AWS cluster.

Note, that small files of the datasets have been created for you to debug and test your code locally on your machine. These datasets can be downloaded from this AWS S3 bucket.
You can use one of the following approaches to load data into your applications:

- Load data from s3 bucket: in this case, you need to set the path to the files in your configuration file to: `s3:///bucketname/pathtoyourfiles`. For example, to load yelp users data into your application directly from s3, you can make changes to configuration file, similar to this one:

  ```
  yelpUserFilePath="s3://bidmtp1data/yelp-data/yelp_academic_dataset_user.json"
  ```

- Copy data into HDFS and then load data from HDFS: you will need to follow these steps:
  - Create the directory structure on local disk at master node of the cluster:

    ```
    mkdir data
    mkdir data/yelp-data
    ```

– Copy file from s3 to local disk at master node of the cluster:

```
aws s3 cp s://bidmtp1data/yelp-data data/yelp-data/ --recursive
```

– Copy data to HDFS:

```
hdfs dfs -put data/yelp-data
```

– Remove the files from local disk since we will not need them any more:

```
rm -r data/yelp-data
```

– In your application, if you want to load files stored on HDFS, you can keep the same paths in the configuration file given to you. For example:

```
yelpUserFilePath="data/yelp-data/yelp_academic_dataset_user.json"
```

Note that the dataset size might be larger than the local disk of the master node and therefore, you will need to copy one file at a time from s3 the HDFS and remove it before copying the following one. Alternatively, you can increase the size of the disk attached to the master node (or attach a new EBS volume).

Part I: Analyzing Airline Delay and Cancellation Data

Input Data

In this part of the project, you analyze a dataset reprsesenting airline delay and cancellation collected during the years 2009-2018. The dataset is divided into several csv files. A file named 20XX.csv represents the data collected in the year 20XX. The schema of the dataset is described at: `https://www.kaggle.com/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018/`.

Part 1.a

The Airline Delay and Cancellation data have several columns that include ″OP_CARRIER″, which is a unique identifier for each airline carrier, and ″CANCELLED″, which is set to ″1.0″ if the flight was canceled and to ″0.0″ if the flight was not canceled. We would like to rank the airline carriers based on the number of times they had canceled flights throughout the years (2009-2018).
We will load the dataset as an RDD and use RDD transformations to analyze the data. First, we need to generate a list of carriers that appears in the dataset. This will require that you complete the code of the following function:

```
def findDistinctAirlineCarriers(
       airlinesData: RDD[FlightDelayCancellationInfo]
       ) : List[String] = ???
```

Second, we will use one of the three following approaches to rank the carriers. The output is pairs of carrier code (″OP_CARRIER″) and the number of flight cancellations for that carrier in the entire input dataset.

Approach 1: Naive Approach
Using this approach, for each carrier in the list of carriers returned by $findDistinctAirlineCarriers$, find the rows in the dataset for that carrier that are marked as "CANCELLED" and return their count. Sort the results in a descending order according to the count of cancellations for each carrier. You will need to complete the code for the following functions:

```
def flightCancellationsForCarrier(
       carrier: String,
       airlineCancellationsRDD : RDD[FlightDelayCancellationInfo]
       ) : Int = ???

def rankingByCounting(
       airlineCancellationsRDD : RDD[FlightDelayCancellationInfo] ,
       carriers : List[String]
       ) : List[(String, Int)] = ???
```

Approach 2: Using an Index
You first need to create an index to map each carrier with all the rows (flights) in the dataset that are related to it and that indicate a flight cancellation. Next, you can use this index to count the entries for each carrier that indicate a flight cancellation. Finally, sort the carriers in a descending order according to the count of flight cancellations for each of them. Note that using an index is useful if it will be for example used several times. You will need to complete the code for the following functions:

```
def generateIndexOfCancellations(
       airlineCancellationsRDD : RDD[FlightDelayCancellationInfo] ,
       carriers : List[String]
       ) : RDD[(String,Iterable[FlightDelayCancellationInfo])] = ???

def rankingUsingIndex(
       airlineCancellationsIndexRDD :
            RDD[(String,Iterable[FlightDelayCancellationInfo])]
       ) : RDD[(String,Int)] = ???
```

Approach 3: Naive ReduceByKey
In this approach, you need to use ReduceByKey to group all the rows in the data by "OP_CARRIER" and count the flight cancellation occurrences for each carrier. Note that you will probably need to first transform the data into a form that is suitable for ReduceByKey. Finally, you will need to sort the carriers in a descending order according to the count of flight cancellations for each of them. You will need to complete the code for the following function:

```
def rankingByReduction(
       airlineCancellationsRDD : RDD[FlightDelayCancellationInfo] ,
       carriers : List[String]
       ) : RDD[(String,Int)] = ???
```

Note that when you run your application, you can enter the approach number (1, 2, or 3) as a command line argument. If no command line argument is entered, then the three approaches will be called.

Part 1.b

Note: We advise you to complete Part 2 before working on this part of the project.

You will need to create a Spark SQL application that analyzes the Airline Delay and Cancellation data and rank carriers according to two features: (1) number of cancellations for each carrier (o/p is expected to be carrier, number of cancellation in all the years 2009-2018); (2) the total flight delays per year per carrier (o/p is expected to be carrier, year, number of cancellation in that year).

Part II: Analyzing Yelp Reviews

Data

In this part of the project, you analyze a dataset that includes yelp reviews. The dataset is divided into several json files. We are interested in these three files: yelp_academic_dataset_review.json, yelp_academic_dataset_business.json, and yelp_academic_dataset_users.json. The schema of the datasets are shown at this link: `https://www.kaggle.com/yelp-dataset/yelp-dataset`.

Requirement

The dataset is loaded as Spark DataFrames. You are required to write the code to implement the following queries on the Spark DataFrames using both SQL statement and DataFrame transformations.

- **Q1:** Analyze yelp_academic_dataset_business.json to find the total number of reviews for all businesses. The output should be in the form of DataFrame with one value representing the count. You will need to write the code for these two functions:

```
def totalReviewsSQL(yelpBusinesses : DataFrame):DataFrame = ???

def totalReviewsbDF(yelpBusinesses : DataFrame):DataFrame = ???
```

- **Q2:** Analyze yelp_academic_dataset_business.json to find all businesses that have received 5 stars and that have been reviewed by 1000 or more users. The output should be in the form of DataFrame of (name, stars, review_count). You will need to write the code for these two functions:

```
def fiveStarBusinessesSQL(yelpBusinesses: DataFrame):DataFrame = ???

def fiveStarBusinessesDF(yelpBusinesses: DataFrame):DataFrame = ???
```

- **Q3:** Analyze yelp_academic_dataset_users.json to find the *influencer* users who have written more than 1000 reviews. The output should be in the form of DataFrame of user_id . You will need to write the code for these two functions:

```
def findInfluencerUserSQL(yelpUsers : DataFrame):DataFrame = ???

def findInfluencerUserDF(yelpUsers : DataFrame):DataFrame = ???
```

- **Q4:** Analyze yelp_academic_dataset_review.json, yelp_academic_dataset_business.json, and a view created from your answer to Q3 to find the businesses that have been reviewed by more than 5 *influencer* users. The output should be in the form of DataFrame of names of businesses that match the described criteria. You will need to write the code for these two functions:

  ```
  def findFamousBusinessesSQL(yelpBusinesses: DataFrame,
         yelpReviews: DataFrame,
         influencerUsers: DataFrame) : DataFrame = ???

  def findFamousBusinessesDF(yelpBusinesses: DataFrame,
         yelpReviews: DataFrame,
         influencerUsersDF: DataFrame): DataFrame = ???
  ```

- **Q5:** Analyze yelp_academic_dataset_review.json and yelp_academic_dataset_users.json to find a descendingly ordered list of users based on their the average star counts given by each of them in all the reviews that they have written. The output should be in the form of DataFrame of (user names and average stars). You will need to write the code for these two functions:

  ```
  def findavgStarsByUserSQL(yelpReviews: DataFrame,
         yelpUsers: DataFrame):DataFrame = ???

  def findavgStarsByUserDF(yelpReviews: DataFrame,
         yelpUsers: DataFrame):DataFrame = ???
  ```

Note that when you run your application, you can enter the query number (1-5) and the approach number (1=SQL, 2=DF) as command line arguments. If you only enter the query number, then both SQL and DF functions will be called. If no command line arguments are entered, then all the queries will be executed.

Running Your Applications

Locally on Your Machine

You can select the file to be run, right click on it and select "run". Alternatively, on your terminal and after you make sure that your current directory is the project, you can type the following to run AirlineDataAnalysisRDD:

```
sbt
sbt:BIDMT_F19_P2> runMain dk.itu.BIDMT.F19.P2.Part1.AirlineDataAnalysisRDD
```

On AWS Cluster

First you need to create a jar file for your project and copy it to the master node of your EMR cluster as shown to you in Exercise 6. Then, you can use spark-submit to execute the code on the cluster.

We are providing a script file "run.sh" that you can use to run the jar file. You need to copy "run.sh" to the master node of the cluster. Then, you need to make sure that it is executable by running this command:

```
chmod u+x run.sh
```

You can now call the script and give it a number that selects the application that you want to run and optionally give it the values for the command line arguments that you would like to pass to your application. For example, to run SQL implementation of query 5 in the YelpAnalysis application, you need to type in the command line:

```
./run.sh 3 5 1
```

For flexibility, you can use a custom configuration file. it is assumed that it is located at the same path as ”run.sh” and it is called ”application.conf”. You can copy the same configuration file provided to you in the project path ”src/main/resources/application.conf”.

Notes

- We will execute your code and compare thee execution time of the submissions of all groups while preparing the feedback. We will potentially publish a scoreboard as part of the feedback. Therefore, try to optimize your code and choose wisely where to persist your RDDs/DFs.
- You will need lectures and exercises of weeks 5-8 to complete this project.
- It is recommended that you work on part 2 before part 1.b.
- You will need to download the datasets and store them in the path: ”data/xxxx” in your project directory (if you are execution the project locally) or on HDFS if you are execution it on the cluster. ”xxxx” is the dataset name: `airline-delay-and-cancellation-data` for the first dataset and yelp-data for the second dataset. (Note: if you store the files at a different location, you will need to change the path in src/main/resources/application.conf)
- Check the configuration file ”src/main/resources/application.conf” for various ways to specify the paths for input files.
- Before preparing your jar file for running your application on the cluster, do not forget to comment this part in your code ”.setMaster(”local[4]”)”
- Note that you will need to choose the right locations of your code to persist the RDDs otherwise the computation might fail or take a very long time to run.
- If any of the applications is taking more than few minutes (¡ 5 minutes) when your run it on the cluster, then there must be something wrong or you need to optimize your implementation.
- Do not forget to remove the output files created when you run AirlineDataAnalysisRDD before your re-run it again, otherwise, an error will be returned.

Deliverable

You need to deliver the following:

- Your code. **Only the scala files.**
- A report that includes the following:
  - A brief description of your solution.
  - A comparison of the execution time performance of the three approaches implemented in part I.
  - A discussion of the type of optimization techniques employed by Spark for your queries in part II. (Hint: You will need to use *explain* or the Spark UI to show the execution plans)
  - A discussion of the effect on performance of using *persist* in your application.
  - An performance analysis showing the effect of increasing the cluster size (number of nodes of the cluster) and the type of instances of the cluster for the three applications.