IS 6493

Homework Exercise 5: Spark DataFrames

(Total: 40 points)

For this homework, you will perform all tasks using Apache Spark in the cloud with Databricks.

A Spark DataFrame is a distributed collection of tabular data grouped into named columns that have a schema for faster and efficient processing. Spark DataFrames are designed for processing large collection of structured or semi-structured data.

A DataFrame is equivalent to a relational table in Spark SQL, and often the same operations can be given as either SQL expressions or PySpark language commands (the latter is more flexible, and what we will learn in this exercise). We will be using Python programming language commands to talk to Spark (PySpark).

In order to learn how to correctly use available methods there is no alternative to the API reference- it is the best resource and one can quickly find out much about the method (parameters and what it returns) by looking up methods in the links below.

While working with DataFrames, we will need the following classes: **DataFrame**, **Data Types**, **Column**, **GroupedData** and **Functions**. Each type of class (object) has its own set of methods.

Please load the relevant URL for each class in a browser window as you work on this exercise (to learn more).

API REFERENCES

Please see the links in the Databricks Notebooks for the week.

There's a lot of functions and methods, you do not need to know them all, only need to know where to look to find the methods when you need them. In this exercise, we will apply what we already know from the in-class lectures and learn a few more methods and apply it to Flight data. I will give you instructions and the methods to use. – you just need to put the code together. This is Learning by doing.

When I say DataFrame or DF in the rest of the document, I am referring to Spark DataFrames.

Please read the two Databricks cloud notebooks: Spark-FileSystem and Spark DF – The Basics and the two related slide sets prior to completing this exercise.

DATA

We will use a dataset that tracks the number of flights between the US and other countries from 2010-2015. The data is split into multiple files, one for each year. The file contains three columns – destination country, origin country, and flight count.

We can ask several interesting questions using this dataset, which will help you practice how to perform exploratory analytics using Spark. Descriptive statistics (and slicing/dicing) allows us to learn more about the phenomenon recorded in the data. For this homework, you will be asked to answer several questions.

This dataset is pre-loaded in the Databricks File System DBFS. It can be found at:

"dbfs:/databricks-datasets/definitive-guide/data/flight-data/csv/ "

SUBMISSION INSTRUCTIONS (PLEASE READ CAREFULLY)

- Create a Databricks notebook and name the file **Hw5_LastName_FirstName**
- Make the first cell a **Markdown cell** with L2 title TASK 0, followed by a L3 sub-title 'Exercise 5 Completed by Your name'
- Use **Markdown** (start cell with %md) to clearly label each task – TASK 1, TASK 2, in L2 headings.
- Use a **Python comment** to indicate each subtask using # 1a #1b …
  - ***Use a different cell for each subtask***.
  - Write all the code for each ***subtask in one cell***.
  - This is really important to do, as it makes grading easier, so please comply.
- Some tasks have multiple outputs. **Every sub-task output should be visible**
  - if you produce more than one output per cell, make sure the outputs asked for are visible in the published notebook (use **print()** if needed when printing out Python values – don't need print when output is a DataFrame).
  - **If we can't see it we can't grade it.**
- Make sure to execute ***Run All*** on your notebook before you export your Notebook, so that all outputs are visible.
  - If you have an error mid-way, *Run All* will not execute cells after the error. You will need to make sure to select *Run all Below* in order to execute later cells (after an error that you can't fix).
- **Submit two things:**
  - Export your notebook as **Hw5_LastName_FirstName.ipynb** and submit it.
  - Publish your notebook and write the URL in the submission comment.

SOME NOTES ON TASKS

- Only perform tasks marked as alphabets a, b, c….
- i, ii are sub-steps – they should be combined to produce the output requested in each a, b, c …
- Please show results in tabular form, unless requested otherwise. Refrain from using DF.***collect()*** as this will load the entire distributed contents into the driver's memory – a very memory expensive task.

CODING EXERCISES

HOUSEKEEPING

TASK 0 (1.5 points)

a. In the first cell, write some markdown as a L2 heading: Task 0
And a L3 sub-heading: **Exercise 5 completed by Your Name**

To enter Markdown, start the cell with **%md** leave a space or hit enter and write markdown.

b. We will need the following import statements
import pyspark.sql.functions as F
from pyspark.sql.types import *

LOADING DATA

TASK 1**:** Display the content of a directory that contains the data (1 point)

The **path** for the data folder is: **"dbfs:/databricks-datasets/definitive-guide/data/flight-data/csv/ "**.

[Please be careful when you copy paste the above line, to check that all chars are included, hyphens often get dropped]

Similar to Unix, you can use either filesystem commands or use method ***ls*** of ***dbutils.fs.*** Please see the associated Databricks notebook on working with FileSystem. You will use both below.

a. View the contents of the data folder using a file system command
   i. For file system commands, designate the cell as ***%fs*** then type ***ls*** and give it an optional argument for the path (otherwise it will print the default current working directory)
   %fs ls path
   ii. Note: you cannot use python comments (# …) inside a cell with magics commands

b. Use the dbutils module and display function to view directory contents as a table
   i. A second approach is to call a the **dbutils** module.
   dbutils.fs is the PySpark module that contains method ***ls()***, so call the ls method as ***dbutils.fs.ls(path)***
   and give it a path you want as a string as argument. ***dbutils.fs.ls('path')***
   ii. Combine the above command with the ***display()*** function to view the results of the above command – ***display()*** is a Databricks Notebook function (not available at the time of writing in other (e.g., Jupyter) Notebooks.

Note the files for flight data available in the directory.

TASK 2: Read and view schema of two .csv files (2 points)

In this task we will read in external files seen above to obtain data for analysis. We will read .csv files in one by one and save each as a DataFrame, and then combine them into one DF. We will do this for only two files here – for years 2015 and 2014.

The command to read/load a .csv file is ***spark.read.csv()*** where
- ***spark*** is the SparkSession instance (automatically available in Databricks)
- ***spark.read*** returns an instance of class **DataFrameReader**

DataFrameReader is the class that has many methods to read different types of data sources, of which .csv() is one
- You can provide several options to ***spark.read.csv().***
  Check out the parameters ***header*** (=False), ***sep*** (= ‘,’) and ***inferSchema*** (=False). Defaults are shown in (=).
- Optional parameters and their values can be provided as follows

spark.read.csv(filepath, option1=value1, option2=value2, …)

There is a second way to read using (use either):
spark.read.format(‘fileformatname’).option(‘paramname’, value)… option(‘paramname’, value).load(‘path’)

a. Read the 2015 data in and save the file as *flight2015DF*
   i. The data has headers, so change the options appropriately (or else you will get errors)
   ii. The default separator is ok
   iii. Keep ***inferSchema*** = False

b. View the schema in tree format using ***printSchema()*** method of Spark DataFrame
   i. observe that everything is read in as a String

c. Now set option/parameter ***inferSchema*** to True, and re-read 2015 data

d. View the schema in tree format using ***printSchema()*** method of Spark DataFrame
   i. Observe that the data type of count is now integer

e. Read in another file, 2014 data, and save it in another DF, *flight2014D*F
   i. Keep option/parameter ***inferSchema = True***

TASK 3: Combine two Dataframes and read in all .csv files (1 point)

Let us learn how to combine the two DataFrames that we created above, and then we will learn how to combine data from multiple input (.csv) files while reading in the data itself to obtain just one DataFrame.

a. First combine the two DFs created in Task 2 for years 2015 and 2014 together using the ***union()*** method of Spark DataFrame.
    i. This method takes another DF as a parameter, and combines together the two DFs by adding new rows.

b. Print the number of rows from combining 2015 and 2104 files
    i. Use the ***count()*** method of DataFrame, which return the number of rows

c. Combine data from all six input .csv files for 2010-2015
Read all *.csv files in the path given above (there are six files) in one step without calling each one separately, while asking Spark to infer the schema, and store them collectively in one DataFrame. Let's call it *flightsAllDF* (notice the 's' in flights)

The same method ***spark.read.csv()*** can be used and the file name is given as ****.csv***
    i. Only the files ending in .csv will be read (if there are other types, they will be ignored)
    ii. Infer the schema and read in header appropriately

TASK 4. Read in all .csv files using a custom schema (3 points)

Read/load all *.csv files by specifying a custom schema instead of inferring it. Inferring is sometimes an expensive process in large datasets, so it is useful to be able to give the schema to ***spark.read*** methods.

a. Create a custom schema as below

```
myschema = StructType( [
        StructField("dest", StringType()),
        StructField("origin", StringType()),
        StructField("count", LongType() ] )
```

You can re-name columns by giving new names in the schema, as we did above. Here we are creating Spark DataTypes **StringType** and **LongType** using their constructor (that is what is being called when you type Classname()).

[A constructor is a special method with the same name as the class that is only called when we want to create a new instance or object of the class. It often accepts some parameters as initialization values or may use default values if no arguments are given.]

Read in all the files again as in task 3c, but this time create and provide *myschema* as the custom schema to ***spark.read.csv()*** using the option, ***schema = myschema.*** Re-save in *flightsAllDF.* In order to only read in csv files don't forget to use *.csv in the path (otherwise data schema will have errors if non-csv are present). Don't forget about headers.

b. Print the schema in tree format using ***printSchema()*** method of Spark DataFrame.
    i. Examine the schema and confirm that the data type of count is different. Also notice that we have given new names to the first two columns – renamed as *dest* and *origin*.

c. Output the count of rows in *flightsAllDF* and the count of distinct rows in *flightsAllDF* and explain the difference
    i. Write a brief comment in a new cell to explain why are the two counts different – and what is missing that if added will reduce these duplicates?

We will refer to this file as *flightsAllDF* and work with this going forth.

TASK 5. View the DF (4 points)

a. Use an action command to view **all** the records of the DataFrame *flightsAllDF* at the Driver node. Caveat: while we would not do this for a very large dataset, it is fine for this task.
    i. The output should be a list of rows.

b. Use three different action commands to view just three rows of the distributed DF
    i. Notice the data type of the output for these different commands – sometimes they return lists, other times a table

c. Print the type of the first row of the DataFrame.
    i. Obtain the first row using the ***first()*** method of Spark DataFrame and the Python ***type()*** function.

DATA PROCESSING

TASK 6. Obtain a column called *filename* and rename the *count* column (4.5 points)

Unfortunately, we do not have a column to indicate the year of the record, which is important. Let us fix this by leveraging the fact that the file name contains the year. Verify this.

We will use Spark DataFrame's methods ***withColumn()*** and ***withColumnRenamed()*** for this task. Please look up the documentation using the link to the DataFrame's API Reference provided above.

a. Create a new DF (say *flightsAllDF_1*) with two changes as described below – rename an existing column 'count' to 'numflights', and add a new column with the filename as values – you should chain them together in the same command.

i. To create a new column by renaming an existing column, you can use the ***withColumnRenamed()*** method of Spark DataFrame. This takes two parameters- the old column name and the new column name.

ii. To create a new column in the DF, use the ***withColumn()*** method of Spark DataFrame. This takes two parameters. The first is the column name as a string. The second is the command to create the values. We want the values to be the filename? How do we retrieve the filename for each row?
    To obtain the filename that the data came from, you can use a function in pyspark.sql.functions called. ***input_file_name()***. This function returns a column of values which are the names of the files that the record/row came from. To call this method, we only need to type ***F.input_file_name()*** since we have imported **pyspark.sql.functions** in our program and stored it with the name **F** (see imports above).

iii. The two methods can be chained together as they both are methods of a Spark DataFrame and each returns a new DataFrame. The final output should be a DataFrame with 4 columns – *dest*, *origin*, *numflights* and *filename*.

b. Repeat task 6a. using ***select()*** method of Spark DataFrame and column expressions
   i. The ***select()*** method takes a list of column expressions, one for each column that you want to include in the output DataFrame.

   ii. The column expression can be,
      i. in the simplest form, just a string representing an existing column name in the DataFrame. To select all existing columns, use '*'
      ii. Or the column expression can be a function or calculation that returns a column object (such as ***F.col('colname')*** or many functions from pyspark.sql.functions, which return columns).

   iii. To rename a column inside ***select()***, you can call ***alias()*** method.
      i. If you refer to a column by just its string name *'colname'*, you cannot then call alias() method on this. ***'colname'.alias()*** will produce an error as *'colname'* is not a column object – it is just a string. To obtain a column object – you would need to first call ***DF['colname']*** or ***DF.colname***, or you can use ***F.col()*** from pyspark.sql.functions.

NOTE: do not use .withColumn() and withColumnRenamed() for 6b.

c. Output 5 records showing **just** the *numflights* and *filename* columns in table form:
   i. Use ***DF.select()*** to select which columns we want from the DataFrame.
   ii. Then, use the ***show()*** action command to view 5 records.

set the optional parameter ***truncate= False*** of show() to be able to see the whole filename.

d. In order to continue using *FlightsAllDF* as our main DF, let us set it equal to the one created above, *FlightsAllDF_1*.
   i. Print out just the column names of *FlightsAllDF*. There should only be four column names (HINT: use a DataFrame attribute)

TASK 7. Add a year column extracted from filename (4 points)

The explanation for this task is a bit long; so please read carefully. Lots of new things to learn.

Having the whole filename in a column is not helpful – the point was to extract the year from it. Let us add a new column called **'year'** with year values as an int, and drop the **'filename'** column from *FlightsAllDF*

a. Add a new column to *flightsAllDF* called ***'year'*** (cast as type integer), and populate it with just the year extracted from the column filename above. Save this is a new DF called *flightsAllDF_2*

   NOTE: (save in a new DF to avoid overwriting file if we make mistakes)

Here is an example filename:

dbfs:/databricks-datasets/definitive-guide/data/flight-data/csv/2010-summary.csv

How should we extract just the year 2010? We will use regular expressions – which are handy tool for specifying text patterns to match (and extract).

Step1. Add a new column

We want all columns from the original DataFrame *FlightsAllDF* and we want to add a new column called **'year'**. For this task, we will use ***DF.select()***

DF.select() is used to create a new DF formed from a (sub)set of columns from a DF. We can also add columns to it by using column_expressions. The syntax is

DF.select('*', column_expression, column_expression …..)

'*' indicates to select and include all existing columns from the existing DF

DF.select(column_expression, column_expression …..)

If you don't want all, specify the column names of the ones you want separated by commas using column expressions. ***column_expression*** is a formula to create a Column, one for each column we want, separated by commas. These can be str of column names or pyspark.sql.functions that produce column of values.

We can use regular expressions to populate a column called ***'year'*** with values extracted from an existing column in *flightsAllDF*- namely, ***'filename'. To apply regular expressions,*** we can use a function in ***pyspark.sql.functions*** called ***regexp_extract(),*** which will extract a substring that matches a specified pattern from a specific column**.** This method will be called as follows:

F. regexp_extract('colnametoextractfrom' , 'regexp', match_group_number)

- ***'colnametoextractfrom'*** will be ***'filename'***
- ***'regexp'*** is a string patterns that specifies that we want to **match a sequence of exactly 4 digits.** (Please see the separate box for regexp below)

In order to extract the matched substring, we use () around the part of the regexp that we want to extract. This is the concept of **match groups**. Here in our case, we want the four digits, so place () around them in the regexp. () for match groups are placed inside the ' '.

- ***match_group_number*** is the index of the () which contains the substring we want.

Sometimes, when we use regexp, we may want to match multiple things and use more than one set of matching parentheses. Each match group is numbered left to right starting at 1. Match group number = 0 represents the whole regexp – everything inside ' '. In this case, since we are only matching one substring, the year, the match_group_number = 1. Caveat: if your regex only contains the pattern for digits, you can use zero index. But often we want to include some surrounding chars to make up a pattern to ensure that the right digits are picked up (when there are several different sets of digits in the source values). If that's the case, then specify index as >0.

F. regexp_extract() returns a ***pyspark.sql.Column*** object to us – which is a new computed column of four-digit year in string form (why String? - regexp returns matched substrings).

We need to do two more things - rename this column and change its data type from String to int.

Step 2: Rename the new column

Rename the newly created column. Once we have a **pyspark.sql.Column** object, we can call its ***.alias()*** method to give it a new name as string

Step 3. Change the data type of the new column

Change its data type using the ***.cast()*** method of **pyspark.sql.Column**. Give it a Spark data type – **IntegerType**. In order to create new integers, we have to call the constructor method of the type using ***IntegerType(),*** or there is a shorthand notation – you can just write ‘int’.

All three steps – compute a new column of values, rename it and cast it to a new type can be chained together inside the DF.select() command. I give you skeleton code below. You may use it to complete the task.

REGULAR EXPRESSIONS

Here, I will give you a quick introduction to regexp. While this is a moderately big topic, you only need to know its basics to solve Task 7.

First, some motivation for using regexp

In this task, what you want to do is extract just the year from a string column, which contains the filename. Here's what a filename looks like:

'dbfs:/databricks-datasets/definitive-guide/data/flight-data/csv/2010-summary.csv'

We want to extract just 2010. How can we do that? solution: use regexp. Regular expressions are a powerful tool available in most programming languages to allow us to work with strings, in particular when we want to extract substrings that match a specified pattern from larger strings. We use regexp to write out the patterns we want to match.

Let's apply regexp to our problem

In TASK 7, the pattern we want is a sequence of 4 digits (0-9).

how can we specify that? for a quick guide you can see here.

https://www.regular-expressions.info/quickstart.html (Links to an external site.)

What you need is to find the sections in the above link titled:

- **Character Classes, Shorthand Character Classes** to see how to specify a digit or 0-9

- **Repetition** to see how to specify that you want 4 characters of a specified character class

- **Grouping and capturing** to see how to then obtain the content of the substring that matches the specified pattern. Essentially you will place parentheses () around the part of the pattern you want to extract - each set of () is known as a match group. Their numbering starts at 1 with the leftmost parenthesis numbered 1. If you don't use match groups, then there is only one default match group numbered at 0 - which is the substring that matches the whole pattern.

EXAMPLES

(Note that patterns are always specified as string using ' ')

If I want to match one digit, we can write the regexp to be **'[0-9]'** or equivalently **'\d'** (using shorthand). If I want to match one capital letter, we can write the regexp to be **'[A-Z]'**, and one lower case letter would be **'[a-z]'**. If we want to match one letter irrespective of case, write **'[A-Za-z]'**

If I want to specify a pattern to match one or more digits, I can write the regexp to be **'[0-9]+'**. To match a digit either exactly once or zero times, we can write '**\d?' or '[0-9]?'.** To match a upper case letter between A-F zero or more times, we can write **'[A-F]*'.**

If I want to match exactly 2 times, we can write **'\d{2}'** or **'[0-9]{2}'**. If I want to match a letter a minimum of two times and a maximum of 3 times, we can write **'[A-Za-z]{2,3}'** (e.g., this last one matches 'dd', 'ZZ', 'eee', 'GGG', but not 'efg' or 'eE').

The above regexp will need to be written as the second parameter of ***F.regexp_extract()*** function. The first and third parameters are discussed above.

```
flightsAllDF_2 = \
    flightsAllDF.select(__________,
                        F.regexp_extract(_________,_______,___)
                                  ,__________ \
                                  ,__________ )
```

```
# Select the columns you want
# Create a new column of extracted string
# Rename the newly created column
# Cast the type of newly created column
```

b. Re-save this new DF *flightsAllDF_2* back in *flightsAllDF*
    i. Drop the column *filename* if you had included it in *flightsAllDF_2.* You can use the
    drop('colname') method of Spark DataFrame if needed.

c. View 5 records in table format showing all columns of *flightsAllDF*
    i. Your DF should now contain only four columns – ***dest*, *origin*, *numflights*, *year*.**

d. View the data types of the columns of *flightsAllDF* using a DataFrame attribute
    i. Ensure that *year* column is an int
    ii. *numflights* column might show up as bigint for long

It is very crucial that you obtain this year column – or you can't perform the rest of the task correctly.

DATA ANALYSIS

For the following tasks, you can **use any methods** to produce the answers if no specific instructions are given. There are many ways to answer the following questions about the data.

TASK 8. Sorting DF (1.5 points)

a. Write code to sort *flightsAllDF* by '***dest***' in ascending order and '***numflights'*** in descending order. View the top 10 records

HINTS.

You can use ***DF.sort()*** or ***DF.orderBy()*** – they are aliases. After selecting the columns, there are many ways to sort.

You may start with a column object obtained from a column expression and call the **pyspark.sql.Column** methods ***.asc()*** and ***.desc()*** OR

Use **pypark.sql.function *.asc('colname')*** and ***.desc('colname')*** which accept string colnames to sort asc or desc

There are others as well as seen in our Notebooks.

TASK 9. Counts of distinct column values (1.5 points)

Calculate counts of distinct values

a. How many distinct destinations are in the dataset?
    i. Write code using a function in the pyspark.sql.functions module – ***countDistinct()***

b. How many distinct combinations of destination and year are there?
    i. Write code using Spark DataFrame methods ***distinct()*** or ***count()***

HINT:

For both tasks, you would use select to create a new DF with only the columns that you want, and then use the function or DF methods mentioned above. Depending on how you write the code, your answer might either be an int or a DF with one column and one row with the count. Either is fine.

TASK 10. Flights from USA to Japan or South Korea (2 points)

Obtain counts of flights to Japan or South Korea as destination

a. How many records of *flightsAllDF* contain either Japan or South Korea as the destination? Answer should be one number (a count as an int).

b. How many total flights (number of flights, not number of observations in data) did United States send to South Korea in 2013? Output the count as a DataFrame (with columns - dest, origin, numflights and year).

HINTS:

For both questions, to select a subset of rows you can use ***.filter()*** or ***.where()*** methods of Spark DataFrame, or indexing ***DF[column condition]*** and Column expressions to perform the comparison task

TASK 11. Produce a bar chart of the domestic flights across years (2.5 points)

We will do a little charting here using the ***display()*** function.

a. Display a bar chart of the number of domestic flights(Y) vs year(X), sorted by year in ascending order.

In our data, domestic flight means that the destination is the same as origin (= USA). Our files only contain domestic flights for the United States, so there are 6 total rows, 1 for each year. We just need to select the appropriate rows from *FlightsAllDF* and sort by *year* column.

Once you have the data as a DataFrame, use the ***display()*** function to obtain a table. Click on the chart option (see little chart symbol at the bottom) to convert it into a bar chart Then choose plot options, and you can select the Aggregation function to apply (e.g., SUM), select the *Keys* (x -axis) and *Values* (y-axis).

This plotting ability is only available using the ***display()*** function – which is a Databricks Notebook convenience function (allows quick plotting without having to use other libraries such as matplotlib).

HINTS.

Select the rows you want first using ***filter()*** or ***where()*** methods of Spark DataFrame, or using indexing ***-DF[column condition]*** and write Column expressions to describe the boolean criteria for the rows. You can chain the ***.sort()*** method of Spark DataFrame to sort the results. Finally call the **display()** function on the resulting DataFrame.

TASK 12. Counts of total flights (2 points)

a. How many total flights (domestic and international) are flown in each of the 6 years? Print the output sorted in descending order of the sum of number of flights column. The Result should be a DataFrame with two columns – *year* and *sum(numflights)*

HINTS.

This is a grouping operation. First use ***.groupby()*** method of Spark DataFrame to group by year, which will give us a **pyspark.sql.GroupedData** object. Then we can call an aggregate method in the ***GroupedData*** class – ***.sum()***. This returns a Grouped DataFrame, with only two columns *year* and the newly calculated column – *sum(numflights)*. Recall that is the name of this column by default. Finally, call ***.sort()*** method of Spark DataFrame to descending sort by the sum of num of flights in each year.

TASK 13. Counts of International flights (6 points)

a. What is the total number of international flights that flew ***to*** the United States in each of the 6 years? Obtain a table with the column of flight count values named as *totalflightsin* instead of the default, and sort the results in ascending order of *totalflightsin.*

Display the results as a DataFrame with two columns – *year* and *totalflightsin*. Save this DF as *inUSADF* as you will need it for part 13.b.

HINTS.

First we want to filter the rows to only keep rows where the ***'dest'*** column IS United States and the ***'origin'*** column is NOT United States. Then apply ***groupby()*** method of Spark DataFrame and group by *year* column, and compute the ***sum()*** of **GroupedData** object over *numflights* column. Then call the ***select()*** method of Spark DataFrame to create a new DF and choose the columns you want in the final DF. Inside ***select()*** you can rename the *sum(numflights)* column using the ***.alias()*** method of a Column object. Instead of ***.select()*** you can also use ***.withColumnRenamed()*** of Spark DataFrame.

b. In what years were there more international flights flying ***to*** the United States than international flights that flew out of the United States (e.g., due to discontinued routes or other reasons)? Answer the question by producing a new DataFrame that contains the following columns and 6 rows, one for each year. Show its contents.

i. *year* – year of data, in ascending order
ii. *totalflightsin* – total number of international flights to the US as destination
iii. *totalflightout* - total number of international flights from the US as origin
iv. *morein* – a boolean that is set to True if totalflightsin > totalflightsout

HINTS.

inUSADF created above in 13a. gives us the international *totalflightsin* for each year.

We need the data on international flights **out** for each year. So, perform the same calculation as above but for total flights **out** of the United States each year and save in a DF called *outUSADF* with only two columns, *year* and *totalflightsout.*

We want to obtain both these columns *totalflightsin* and *totalflightout* in the same DataFrame. To do this, join two DataFrames *inUSADF* and *outUSADF* by calling the ***join()*** method of Spark DataFrame using the common column ***on = 'year'*** and ***how = 'inner'***.

After joining, you need to create a new column – *morein,* using column expressions to create its values – True or False, depending on whether *totalflightsin* > *totalflightsout*. You can either use ***DF.select()*** or ***DF.withColumn()*** and don't forget to rename the column using the ***.alias()*** method of the Column object or using ***withColumnRenamed()*** method of Spark DataFrame.

TASK 14. Aggregate statistics (3.5 points)

a. For the rows that contain United States as destination but NOT as origin (i.e. international flights into the USA) compute and print the following summary statistics for the column *numflights* – sum, avg, max - for each of the 6 years. The output should have columns named, *year*, *sumflightstoUS*, *avgflightstoUS*, *maxflightstoUS***.** Ascending sort the DataFrame by *year*.

HINTS.

First filter the rows that you want to keep, i.e. ***dest*** is the United States and the *origin* is not United States Then group by *year* column. You can use ***.agg()*** method of the ***pyspark.sql.GroupedData*** class to call summary functions on columns of the DF

DF.groupby(col1, col2, …).agg(aggfunction(col1), aggfunction(col2),……)

Inside ***.agg()*** , we can specify built-in aggregation functions that are found in ***pyspark.sql.functions*** module. Finally, sort by *year* in ascending order.