**SETUP**

For this report, I was given the Databricks community edition to work. With this, I faced some certain issues at some point with creating clusters, having commands run for a lot of minutes etc but I was able to use it to get it done. Onto the datasets, I was given the CORD 19 dataset and specifically the 2020\_07\_01 version to work with and also the journalrank data from scimago.

The CORD19(Covid19 Open Research Dataset) is a collection of academic papers and covid19 related research

This report contains the 3 required implementations using DataFrames, RDDs and Hive queries. The Hive implementation was sort of he toughest to deal with as I had troubles getting the ADD JAR to work till I had to download from the repository and uploading it manually to the cluster to get it working. I highlighted the various functions I used in red, so it would be spotted easily.

At the end of the report, I have shortly discussed the use purpose of the CORD19 dataset

**DATA LOADING**

Using the shell script ‘%sh’ along with ‘wget’, I was able to download both files required for the purpose of this project. I could have use ‘curl’ as well but I chose to go with the ‘wget’. I placed the accurate 2020-07-01 date as requested by the assignment brief and ended it with the metadata.csv. Also, the journalrank file had to be downloaded with an xls format which I did. After checking around databricks using the ‘cd’ command, I noticed both files were downloaded to the ‘/databricks/driver/’ folder which I then used ‘ls’ command to make sure it was there.

A screenshot of a computer

Description automatically generated with medium confidence

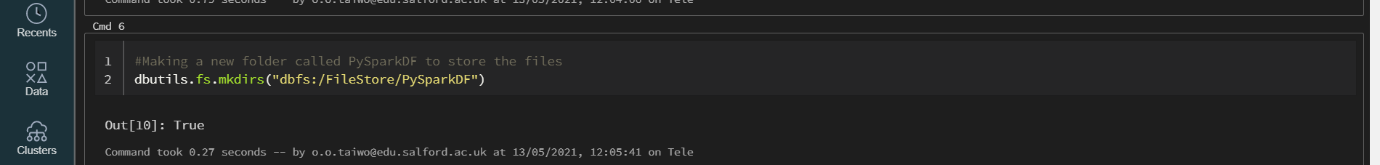
As the assignment brief suggested to rename the Metadata.csv, I used the dbutils.fs.mv command to rename as I tried the regular ‘mv’ command but did not work.

Text

Description automatically generated

**PYSPARK DF**

I created a new folder named ‘PySparkDF’ specifically for the DataFrames notebook, using dbutils.fs.mkdirs, to avoid mistakes.



After this, I was good to go with starting the DataFrames implementation. Now, I copied both files from the ‘/databricks/driver/’ to the initially created PySparkDF folder. I listed the files in the PySparkDF folder to make sure the files I copied were in there and was successful.

Text

Description automatically generated

Using spark.read.options, I was able to convert the journalrank file into DataFrames with a delimiter ‘;’, specifying its location correctly with the code. The ‘inferschema’ function was used to assign appropriate datatypes to help with the implementation and a ‘header=true’ meaning the first line is the header. With this same method, I was able to convert the metadata file as well into a dataframe. Here, I employed the ‘escape’ function to ignore ‘\\’ and ‘”’ so as to give smoother results while the ‘multiline’ function helps to process the CSV file with values in rows scattered across multiple lines After this, I was able to show both files using the ‘show(10)’ to display the first 10 results.

Text

Description automatically generated

Text

Description automatically generated

1. To answer the first question for the top 5 most common journals with their frequency, I filtered out the journal rows that were NULLed, then employed a groupby function to group the journals. As well as using a Count function to get the frequency, I had to order it by the top 5 to arrive at my results.

Text

Description automatically generated

1. For the top 5 average abstract lengths per journal, I had to use the ‘import function’ then made a CovidDF using the initial Metadata\_2020\_07\_01DF. Going further, I specified the ‘abstract’ column as requested, grouped it by the journal column then the ‘F’ function to count the average. The ‘withcolumn’ function was used to create a new column named ‘wordcount’, while the ‘split’ was used to split by white spaces into a list and the ‘size’ function was used to count all the elements in the list. I also employed the use of ‘average’ as my alias along with the ‘ascending=false’ to order it in a descending order. Also using the ‘show(5) at the end, I displayed the accurate results.

Text

Description automatically generated

1. Using the same ‘covidDF’ from the second question, I was able to get the 5 papers with the highest numbers of authors. The ‘withcolumn’ was used again for another ‘wordcount’ column as well as the ‘size’ function and ‘split’ functions. Selecting ‘title’ and ‘authors’ as requested, I was able to show this data accurately also utilizing the ‘ascending =false’.

Text

Description automatically generated

1. For the next question to find the 5 most prolific authors and number of papers they have contributed to, I had to implement the ‘import pyspark.sql.functions as f’, I also used the metadata\_2020\_07\_01DF and had to use the ‘explode’ function, which was used to bring out the names of the authors after the split, with the alias ‘Authors’ for the ‘authors’ column. Further used a ‘count’ as usual and an ‘orderby’ with the same old ‘ascending=false’ to arrive at my results.

Text

Description automatically generated

1. To list the 5 authors with the top author H index values, I had to implement the ‘import pyspark.sql.functions as f’ once more. I had to make use of the ‘journal rank’ table finally. With the ‘title’ column being common to both, I was able to join my metadata\_2020\_07\_01DF with the initially created journalrankDF using an inner join. Moving on, I used the ‘select’ command to show the ‘authors’ and ‘H index’ with the ‘show(5)’

Text

Description automatically generated

Text

Description automatically generated

1. For Plotting the number of papers per month since 2020-01, I first created a CovidDF dataframe from the metadata\_2020\_07\_01DF with the ‘publish\_time’ and ‘title’ column with data containing ‘2020’. I then imported pyspark.sql functions. With this, I created a ‘countmonth’ from the CovidDF, aliasing ‘month’ and grouping it by month, as well as using ‘withcolumnrenamed’ to change names to a ‘count’ and ‘total’ and ordering in a descending order. I then made a ResultDF using the ‘countmonth’ and the cast function to convert the datatype from string to timestamp. From this, I created a FurtherResultDF with the ‘F Function’ imported after which I was able to filter the months to display results after ‘2019-12’.

Text

Description automatically generated

Text

Description automatically generated

I then made an RDD from the results to display months after ‘2019-12’. Next step, we append data into empty ‘month’ and ‘total’ tables then run a Print command to view the results. From this, I imported pyplot to enable me plot a graph for the required output.

Graphical user interface, text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

**PYsparkRDD**

As my general rule, I made sure to check if my required files were in the ‘/databricks/driver/’ so as to know if I’d need to re-download. Seeing as it was there, I proceeded to the PySpark RDD notebook. I created another folder named ‘PySparkRDD’ file to have a clear path which I would follow. After this, I copied the two required files into it as well.

Text

Description automatically generated

When this was done, I listed the files in the PySparkRDD folder to make sure they were right where I wanted them. Using sprak.read.csv, I was able to convert the metadata\_2020\_07\_01 csv file into an a dataframe first, specifying the correct file path. And with the printschema function, I was able to list the columns along with their datatypes and ‘nullable’ feature.

Text

Description automatically generated

After this, I converted the data frame into an RDD using ‘Metadata\_2020\_07\_01RDD = Metadata\_2020\_07\_01DF.rdd’. Also utilizing Python’s ‘for row’ function, I added the ‘take(5)’ to list the first five results with the ‘print’ function.

Graphical user interface, application

Description automatically generated

1. For the 5 most common journals and their frequencies, the ‘map’, ‘lambda’, ‘reducebykey’ which acts as a grouping function for RDDs and ‘sortbykey’ functions relative to RDDs come into play. ‘Fields[11]’ stands for the Journal column which we would be making use of. Using our good old friend ‘take(5)’ from the DataFrame notebook, I listed the results and crosschecked with my results from the DataFrame results. I found them to be equal.

Text

Description automatically generated

1. Using the same metadata2020\_07\_01RDD, I found the top 5 average abstract lengths per journal. Here, the strip to remove white spaces, split, round(sum(x)) to round the sum to a 1 decimal place, len(x), came to play to help with getting the results. ‘Fields[8]’ standing for the Abstract column, Not forgetting the ‘ascending = false’ function being used to order it in a descending manner.

Text

Description automatically generated

1. Here, using the same metadata\_2020\_07\_01, we make use of the ‘fields[3] and fields[10], and the .map which more or less acts like a filter and was split by a ‘;’. Sortby applies here as well and we display results in a descending manner as requested and we get our answers which match with the DF results.

Text

Description automatically generated

1. Here, we are to get the top 5 most prolific authors and the number of papers with ‘fields[10] representing the ‘authors’ column. ‘flatmap’ here does the same as the ‘explode’ function that was used in the DataFrame.

A screenshot of a computer

Description automatically generated

1. We first create a JournalRankDF, specifying the correct location of our files. From this, we create an RDD and print it out to confirm. After this, I joined both JournalRank and metadata\_2020\_07\_01 to have a single RDD. X[2] and x[7] signifies what we want to be shown from the journalrank, while col[11] and col[10] is what we want to be shown from the metadata\_2020\_07\_01 then display the first 5 results. On the next command line, we run the filter, ‘sortby’, ‘reducebykey’ functions to get the necessary results which correlates with our results from the DataFrame implementation to list the 5 people with the top author H index values.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**HIVEQL**

To get my Hive Notebook started, I had to navigate to the site <https://mvnrepository.com/artifact/org.apache.hive/hive-serde/3.1.2> to download the JAR file locally. After which I went back to my Cluster on databricks, then navigated to the Libraries folder then uploaded the JAR file and restarted my cluster.

Graphical user interface, text, application, email

Description automatically generated

The very first code was to add the JAR file to my notebook with the ‘ADD JAR’ command. After this, I created two new folders, one named ‘HiveMetadata’ which was to contain the downloaded metadata.csv file alone and another named ‘Hive’ to house the journalrank table. To avoid any errors, I added both shell scripts to download both files again, renamed the metadata.csv to metadata\_2020\_07\_01, I then copied both files into their respective folders. After this, I listed the files in the ‘HiveMetadata’ to make sure the copied files were right where I wanted them.

A screenshot of a computer

Description automatically generated

Text

Description automatically generated

My next step was to rename the metadata file from metadata\_2020\_07\_01 to meteadata\_2020\_07\_01csv as I realized creating the table was not working with that name. Using the standard table creation format of ‘if not exists, I created the table named ‘metadata\_2020\_07\_01csv’ with all datatypes as strings and ROW FORMAT using opencsv SERDE to read the data. ‘separatorchar’ indicates there are separations by commas while ‘quotechar’ indicates quotes are by double quotations and any commas within are not to be split and the ‘escapechar’ is used to ignore all ‘\\’ wherever it is found. Location specified is the /FileStore/HiveMetadata/ which houses the metadata file alone.

Text

Description automatically generated

The next step was to check if the table had correctly been loaded using the ‘SELECT \*’ command to display all the columns in the table. For the next step, I created a JournalRank table then loaded the downloaded journalrank, specifying the correct file location, first row being the header as well. I also ran a ‘SELECT \*’ command to display all columns in the table to make sure it was loaded properly.

A screenshot of a computer

Description automatically generated

1. To get the journal with the highest frequencies, I ran a SELECT command specifying the journal column and a count(journal) column aliased as Count with a WHERE clause to filter out the rows where the journal column was empty. As it was an aggregate function, I was able to Group By journals the ORDER BY in a descending order and a LIMIT 5, to limit the results to the Top 5.

A screenshot of a computer

Description automatically generated with medium confidence

1. Also to get the highest average length, I ran a SELECT command specifying the journal column and the abstract column with the AVG function to calculate the average and aliased it as Average and then filtered out the rows for abstract and journal WHERE both are empty then GROUP BY journal. Within this, I splitted by white spaces using the Split function and the Size function to count the elements within the array and I was able to get the accurate results as I did with the dataframes and RDD.

A screenshot of a computer

Description automatically generated with medium confidence

1. To get the titles of papers with highest number of authors, I made use of the ‘title’ column and a ‘count(wordcount) aliased as ‘count’. I also made us of LATERAL VIEW explode which does same as the ‘explode’ from DFs and RDDs which explodes elements within the array into rows. After this, grouping was done by the Title column and ordered by the count(wordcount) column. The results were same as that of the other notebooks.

A screenshot of a computer

Description automatically generated with medium confidence

1. To find the top 5 most prolific authors, I made use of a subquery and named it Views, which I used to call the Authors column. Within this subquery, I made use of the Explode function on the Authors column once more which was split by ‘;’ and grouped by the derived Authors column. Outside the subquery, there was the Trim function which as usual was to trim away white spaces. My results were accurate as with the other notebooks implemented.

Text

Description automatically generated

1. Finding the top H-index values, there had to be a join just as the DF and RDD notebooks, so it was joined here on the Title column. I selected the H-Index column and aliased it as Sum, the author names was as well trimmed and aliased as Authors. The journalrank was aliased as jr1. With the lateral view explode, there was a split function as earlier done which was aliased as 'names’ aforementioned above then grouped by the author.

Text

Description automatically generated

**AWS IMPLEMENTATION**

Starting the AWS, I logged in to my Modules section, started the Lab 2 which I was going to use for the purpose of this implementation.

Graphical user interface, application

Description automatically generated

After this, I went into S3 to create a new bucket and in this bucket, I created two new folders called the metadata and journalrank. I had already downloaded both the metadata.csv and journalrank tables into my desktop locally. I went into my new bucket, and uploaded the metadata into metadata and scimago into the journalrank folder respectively.

Graphical user interface, text, application, email

Description automatically generated

Going back to launch Athena, I was asked to query the bucket location which I copied the S3 URI of my folder and pasted it in. This was done after several attempts and errors.

Onto loading the data into Athena query pad, I created the format for the columns and datatypes and also specifying the location of my S3 URI.

Graphical user interface, text, application

Description automatically generated

The loading was successful and I called it with the SELECT \* function, which returned this:

A screenshot of a computer

Description automatically generated with medium confidence

The next table to be created is the scimagojr, which I did almost the same way as the metadata.

Text, application

Description automatically generated

I then used the SELECT \* command to call all the tables to check if it had been created.

A computer screen capture

Description automatically generated with medium confidence

Onto the problem statements, I noticed a few errors here and there in the data and this was due to the fact that there were some quotation marks and commas with, due to improper cleaning of the data.

1. A screenshot of a computer

   Description automatically generated

A computer screen shot

Description automatically generated with low confidence

1. I was able to get the accurate results for the highest average lengths.

A picture containing text, screenshot, monitor, computer

Description automatically generated

A computer screen capture

Description automatically generated with medium confidence

1. I was able to get the correct results for the titles of 5 papers with highest number of authors and had to apply the DISTINCT function as the first result appeared twice.

Graphical user interface, text, application, email

Description automatically generated

**RESULT**

1. Looking at the result from Problem Statement 1, I was able to see that PLoS One had the highest frequency with 1781, with bioRxiv at 1669 coming second. BMJ was third with 1652, Journal of virology with 1620 and lastly , Lancet with 986.

Table

Description automatically generated

1. Absolute Neurocritical Cre Review had the highest average length of words with 2232, World Allergy Organ J came second with 1667, Endocrine, metabolic and immune disorders drug targets was third with 1505, National Toxicology program technical report series was fourth with 1301 and Tob Induc Dis was fifth with 969.

A screenshot of a computer

Description automatically generated with low confidence

Graphical user interface, text

Description automatically generated

1. Table

   Description automatically generated
2. Table

   Description automatically generated

**DISCUSSION ON WHY ANALYSIS IS BEING PERFORMED**

The CORD19 dataset is a joint effort by mainly the Allen Institute for AI (AI2) and The White House Office of Science and Technology Policy (OSTP) and contains several publications related to the coronavirus, including SARS and MERS. The dataset aims to help the AI and text mining community as a research dataset with its data cleaning process, as well as treatment identification for the coronavirus.