Group: LSC Group 25

Members: Kozy-Korpesh Tolep (S5302354) and Urwa Fatima (S5156692)

Motivation:

The amount of data has risen exponentially in the last couple of years, distributed system technologies like Spark are overcoming the challenges of handling and storing data with efficient and optimized APIs. Therefore this lab project is developed to learn more about Spark SQL Dataframe API that handles advanced and complex scenarios for analyzing data.

Objective:

The objective is to implement complex transformations, selections, and extract information from a large [dataset]() of BigMac Burger prices over several years [1]. This dataset has 812 rows and 14 columns.

Following questions are addressed:

1. Find top three countries with most expensive BigMac in 2011
2. Find top 20 countries with max price for BigMac over the years
3. Find the difference between maximum value and minimum value for dollar exchange for each country and sort by difference in descending order
4. Show average local price of Bigmac for each year and find the difference with the previous year's average local price and sort difference in descending order
5. For each country show largest increase in dollar exchange rate(dollar_ex) comparing month of each year and sort in descending order
6. For each year give a rank for countries average dollar_price for BigMac.

API:

A DataFrame is a distributed collection of data with well defined rows and columns which makes it conceptually equivalent to a table in a relational database or a dataframe in R or Python. It differs in executional effeciency because of the SQL-like optimizer called Catalyst working behind the scenes. DataFrame is immutable and it is structured to have lazily evaluated plans that specify what operations to apply to data in order to generate some output.

Sources of data in DataFrames: DataFrames can be constructed from a wide array of sources such as structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, Python, and R.

The DataFrame API can act as a distributed SQL query engine as the input data can be queried by using ad-hoc methods and an SQL-like language for structured data manipulation. Spark has mainly two types of method which can be peformed on DataFrame. These methods also refered as operations are either Action or Transformation

Transformation: The Spark operations that reads a DataFrame, manipulates some of the columns, and returns another DataFrame are considered transformational operations. Functions like select(), filter() and groupBy() are example of transfrormation

Action: The Spark operations that either returns a result or writes to the disc are considered as action function for example, count() and collect().

For a complete list of the types of operations that can be performed on a DataFrame can be referred through the API Documentation.

As DataFrames has the feature of lazily evaluation. This means when Spark transforms data, it does not immediately compute the transformation but plans on how to compute later. When actions such as collect() are explicitly called, then computation starts.

Difference between RDD vs Dataframes vs Datasets:

This table shows the similarities and differences of these Spark data manupilation APIs [3].

RDD	Dataframes	Datasets
Fault Tolerant	Fault Tolerant	Fault Tolerant
Distributed	Distributed	Distributed
Immutable	Immutable	Immutable
No Schema	Schema	Schema
Slow on Non-JVM languages	Faster	Faster
No Execution Optimzation	Catalyst Optimization	Optimization
[illegible]	evel	High Level
No SQL Support	SQL Support	SQL Support
Type Safe	No Type Safe	Type Safe
Syntax Error Detected at Compile time	Syntax Error Detected at Compile time	Syntax Error Detected at Compile time
Analysis Error at Compile Time	Analysis Error Detected at Run Time	Analysis Error at Compile Time
JAVA, SCALA, Python, R	JAVA, SCALA, Python, R	JAVA, SCALA


```
#run this block if the pyspark library is not installed
!pip install pyspark
```

```
Collecting pyspark
  Downloading pyspark-3.2.0.tar.gz (281.3 MB)
```

```
     |████████████████████████████████| 281.3 MB 35 kB/s
Collecting py4j==0.10.9.2
  Downloading py4j-0.10.9.2-py2.py3-none-any.whl (198 kB)
     |████████████████████████████████| 198 kB 62.3 MB/s
Building wheels for collected packages: pyspark
  Building wheel for pyspark (setup.py) ... done
  Created wheel for pyspark: filename=pyspark-3.2.0-py2.py3-none-any.whl size=281805911
  Stored in directory: /root/.cache/pip/wheels/0b/de/d2/9be5d59d7331c6c2a7c1b6d1a4f463c
Successfully built pyspark
Installing collected packages: py4j, pyspark
Successfully installed py4j-0.10.9.2 pyspark-3.2.0
```

```
# installation required to run the session
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql import functions as F
from pyspark.sql.window import Window
import pyspark.sql.functions as func
import time
```

```
#The entry point into all functionality in Spark is the SparkSession class.
#To create a basic SparkSession, just use SparkSession.builder()

# Initialize SparkSession
spark = SparkSession \
    .builder \
    .appName("Group_25") \
    .getOrCreate()
```

```
# Uncomment the datasetPath variable when running the file on front-end server
# datasetPath  = "hdfs:/user/user_lsc_25/BigMac.csv"

# Using this path for the Colab notebook
datasetPath  = "/content/big-mac-adjusted-index.csv"
```

Saved successfully!

```
from google.colab import files
uploaded = files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving big-mac-adjusted-index.csv to big-mac-adjusted-index.csv
```

```
datasetPath = 'big-mac-adjusted-index.csv'
```

```
# Summary of schema and the data in the CSV file of BigMac Burgers Dataset
datasetDF = spark.read.csv(datasetPath, header=True, inferSchema=True)
datasetDF.printSchema()
```

```
root
 |-- date: string (nullable = true)
 |-- iso_a3: string (nullable = true)
 |-- currency_code: string (nullable = true)
 |-- name: string (nullable = true)
 |-- local_price: double (nullable = true)
 |-- dollar_ex: double (nullable = true)
 |-- dollar_price: double (nullable = true)
 |-- GDP_dollar: double (nullable = true)
 |-- adj_price: double (nullable = true)
 |-- USD: double (nullable = true)
 |-- EUR: double (nullable = true)
 |-- GBP: double (nullable = true)
 |-- JPY: double (nullable = true)
 |-- CNY: double (nullable = true)
```

```
print(datasetDF.count(), len(datasetDF.columns))
```

```
812 14
```

Question 1:

Find top three countries with most expensive BigMac in 2011.

Solution:

First we need to filter and leave only records which were made in 2011. However, the date column as it's in the format - 'yyyy-mm-dd', we need to use date() function to obtain a year from there. After that, we select only the name of the country and sort descending by 'dollar_price' and show top 3.

```
# select() returns a new dataframe with the provided column name and where() filters the data
# sort() the function aranges data in the column specified in ascending order by default.

data = datasetDF.where(year('date') == 2011).select('name').sort(desc('dollar_price'))
```

Saved successfully!

```
+-----------+
|       name|
+-----------+
|     Norway|
|Switzerland|
|     Sweden|
+-----------+
only showing top 3 rows

CPU times: user 16.2 ms, sys: 2.42 ms, total: 18.7 ms
Wall time: 1.05 s
```

Question 2:

Find average dollar_price for BigMac for each year and for all countries and sort by year.

Solution:

To find average 'dollar_price' over the years for each country we use the aggregation function avg() from pyspark.sql and group records by 'year'. After that we sort data by 'year' column.

```
# selectExpr() is a variation of select() that projects a set of SQL expression
# .agg() function compute aggregates and return dataframe as a result. The available aggregat

data = datasetDF.selectExpr('year(date) as year', 'dollar_price').groupBy('year').agg(F.avg('
data.show()
```

```
+----+------------------+
|year|         avg_price|
+----+------------------+
|2011|3.9364215199569053|
|2012|  3.64763559762034|
|2013|3.8091063435738315|
|2014| 3.836926679387834|
|2015|3.5356402978675883|
|2016| 3.373439669452451|
|2017| 3.541480205052853|
|2018|3.7087346670848276|
|2019|  3.60018134836105|
|2020| 3.675629022124684|
|2021|3.7285290062520344|
+----+------------------+

CPU times: user 28.4 ms, sys: 1.75 ms, total: 30.1 ms
Wall time: 2.27 s
```

Question 3:

Saved successfully! ✕

value and minimum value for dollar exchange for each

ending order

Solution:

For solving this question we need to get two aggregation functions from one column. That is why we imported pyspark.sql to use max() and min() aggregation functions to one column. After that, we need alias function to name the new columns created by aggregation fuctions. We need to change it from 'max(dollar_ex)' and 'min(dollar_ex)' because when we will use raw sql 'max(dollar_ex)' column can be interpreted by selectExpr() function as a new max() aggregation function. After naming two columns obtained from aggregation functions we use selectExpr() function and subtract minimum of dollar exchange from maximum and sort by their difference.

```
data = datasetDF.groupBy('name').agg(F.max('dollar_ex').alias('max_dollar_ex'), F.min('dollar
data = data.selectExpr('name', 'max_dollar_ex - min_dollar_ex as difference_of_dollar_ex').sc
data.show()
```

```
+--------------+----------------------+
|          name|difference_of_dollar_ex|
+--------------+----------------------+
|     Indonesia|                5994.5|
|      Colombia|               2071.61|
|         Chile|     324.9500000000005|
|   South Korea|     177.2000000000005|
|       Hungary|     123.3055500000001|
|     Argentina|     92.20124999999999|
|      Pakistan|                80.155|
|         Japan|                47.015|
|        Russia|               46.9025|
|       Vietnam|                  35.5|
|         India|    30.797500000000007|
|   Philippines|                12.887|
|         Egypt|                12.808|
|        Mexico|              10.76525|
|     Sri Lanka|                  10.5|
|  South Africa|     9.903649999999999|
|Czech Republic|     8.789850000000001|
|        Turkey|                6.8344|
|      Thailand|     6.469999999999999|
|        Taiwan|    5.2514999999999965|
+--------------+----------------------+
only showing top 20 rows

CPU times: user 21.3 ms, sys: 0 ns, total: 21.3 ms
Wall time: 1.24 s
```

Question 4:

Show average local price of Bigmac for each year and find the difference with the previous year's average local price and sort difference in descending order

Saved successfully!

Firstly, we need to find the average local price for each year as there are two measurements in one year. Then we need to use the window function and partition data by 'name' of the country and use function func.lag() to get the previous value of the 'avg_local_price' column. After that as in the previous solution we find difference between each year and previous one and sort by it.

```
#To perform an operation on a group first,
#we need to partition the data using Window.partitionBy() ,
#and for row number and rank function we need to additionally order by on partition data usir

window = Window.partitionBy('name').orderBy('name')
```

```
data = datasetDF.selectExpr('name', 'local_price', 'year(date) as year').groupBy('name', 'yea
data = data.withColumn('prev_year_local_price', func.lag('avg_local_price').over(window))
data = data.selectExpr('name', 'year', 'avg_local_price', 'avg_local_price - prev_year_local_

data.show()
```

```
+-----------+----+---------------+-----------------------------------------+
|       name|year|avg_local_price|difference_local_price_from_previous_year|
+-----------+----+---------------+-----------------------------------------+
|  Indonesia|2013|        27939.0|                                   4572.0|
|  Indonesia|2018|        33625.0|                                   2062.0|
|  Indonesia|2016|        30750.0|                                   1530.5|
|   Colombia|2017|         9900.0|                                   1500.0|
|   Colombia|2018|        11400.0|                                   1500.0|
|  Indonesia|2015|        29219.5|                                   1280.5|
|   Colombia|2021|        12950.0|                                   1050.0|
|  Indonesia|2020|        33500.0|                                   1000.0|
|  Indonesia|2012|        23367.0|                                    833.0|
|  Indonesia|2017|        31563.0|                                    813.0|
|  Indonesia|2021|        34000.0|                                    500.0|
|   Colombia|2016|         8400.0|                                    500.0|
|   Colombia|2019|        11900.0|                                    500.0|
|South Korea|2015|         4200.0|                                    300.0|
|      Chile|2017|         2500.0|                                    300.0|
|      Chile|2021|         2965.0|                                    275.0|
|      Chile|2012|         2050.0|                                    200.0|
|South Korea|2016|         4350.0|                                    150.0|
|  Argentina|2021|          350.0|                                    139.5|
|      Chile|2018|         2620.0|                                    120.0|
+-----------+----+---------------+-----------------------------------------+
only showing top 20 rows

CPU times: user 29.5 ms, sys: 4.7 ms, total: 34.2 ms
Wall time: 1.15 s
```

Question 5:

For each country show largest increase in dollar exchange rate (dollar_ex) comparing month of each year and sort in descending order

Saved successfully!

To solve this question, we again use window() and func.lag() to partition data by 'year' and 'name' and to get the previous value of dollar exchange for each raw. This time we need to partition by name and year as we need to find previous values for each month separately. For records start from 2011 and for measurements that were made in the first month of each year we get 'NULL' value for func.lag() function as with partitions year and name they don't have dollar_ex for previous raws. Thus, we need to filter resulted data and get rid of rows with null values for the 'previous_dollar_ex' column. Next, we need to find the difference between column 'dollar_ex' with the 'previous_dollar_ex' column. Finally, need to find maximum of their difference grouping by name.

```
window = Window.partitionBy('year', 'name').orderBy('year')
data = datasetDF.selectExpr('name', 'dollar_ex', 'year(date) as year', 'month(date) as month'
data = data.filter('month != 1 and year != 2011').selectExpr('name', 'year', 'dollar_ex', 'ir
data = data.groupBy('name').agg(F.max('difference_dollar_ex')).sort(desc('max(difference_doll
data.show()
```

```
+--------------+------------------------+
|          name|max(difference_dollar_ex)|
+--------------+------------------------+
|     Indonesia|                  1001.0|
|      Colombia|       381.6100000000001|
|   South Korea|      61.950000000000045|
|         Chile|       45.79500000000007|
|       Hungary|      24.288050000000027|
|      Pakistan|      18.569999999999993|
|     Argentina|      11.175049999999999|
|     Sri Lanka|                    10.5|
|        Russia|       9.152499999999996|
|         Japan|       9.040000000000006|
|         India|       6.584999999999994|
|    Costa Rica|      3.9700000000000273|
|        Mexico|       3.615500000000001|
|   Philippines|      2.8399999999999963|
|      Thailand|      2.6800000000000033|
|  South Africa|      2.2814999999999994|
|       Uruguay|      1.5050000000000026|
|Czech Republic|      1.3177500000000002|
|        Brazil|                 1.19855|
|        Turkey|      1.0867499999999994|
+--------------+------------------------+
only showing top 20 rows

CPU times: user 25.5 ms, sys: 5.17 ms, total: 30.7 ms
Wall time: 1.23 s
```

Question 6:

Saved successfully! average dollar_price for BigMac.

Solution:

This question can be solved by rank() function which helps to identify rank of the value in the partition. So, as we have two measurements for each year, we need to find average for each year. Then we use window function and partition data by name and order by 'avg_dollar_price' column for each year and apply rank() fuction. Finally, we need to order data by name and rank

```
window = Window.partitionBy('name').orderBy('avg_dollar_price')
data = datasetDF.selectExpr('name', 'dollar_price', 'year(date) as year').groupBy('name','yea
data = data.withColumn('rank', rank().over(window)).sort('name','rank')
```

```
data.show()
```

```
+---------+----+-----------------+----+
|     name|year|  avg_dollar_price|rank|
+---------+----+-----------------+----+
|Argentina|2019|2.4364536274995503|   1|
|Argentina|2014|2.8024533848179196|   2|
|Argentina|2016|2.8687715986163003|   3|
|Argentina|2015|3.1585833099710303|   4|
|Argentina|2020| 3.178059921365675|   5|
|Argentina|2018| 3.332767902579255|   6|
|Argentina|2017| 3.796962230447655|   7|
|Argentina|2021| 3.846425460025445|   8|
|Argentina|2013| 3.848925979818775|   9|
|Argentina|2012| 4.398784797986675|  10|
|Argentina|2011|  4.83968542044767|  11|
|Australia|2016| 4.024196250690875|   1|
|Australia|2015| 4.120484999455475|   2|
|Australia|2019| 4.306075000000005|   3|
|Australia|2017|  4.40156750080276|   4|
|Australia|2020|  4.51479750000001|   5|
|Australia|2018| 4.610776249831865|   6|
|Australia|2014|  4.64384374945093|   7|
|Australia|2013|  4.76015962477957|   8|
|Australia|2012| 4.808357999562684|   9|
+---------+----+-----------------+----+
only showing top 20 rows
```

Execution Time:

Question	cluster,sec	local,sec
1	7.91	7.21
2	11.31	10.96
3	11.25	10.9
4	17.36	21.6
[illegible]	14.36	16.36
[illegible]	12.63	16.23
all together	26.677	43.617

Enter to this directory path:

user_lsc_25/project/project_1.py

For runing the file locally use:

spark-submit --master local project_1.py

For running the file through yarn:

spark-submit --master yarn project_1.py

Insight:

Running the project file in the local mode was faster for question 1, 2 and 3, which has simple queries with some groupby, filter and aggregation functions. We assume that clustering is slower because it spends some time for scheduling and assigning task to each machine. Local mode right away working on the code. Starting from the 4th Question, we have started using partitioning for solutions and we get more complex queries. As a result, we get more faster results in yarn mode as it gives opportunity to run processes in parallel. Running all question implementations is almost two times faster in cluster mode because it distributes all the task to clusters and run it in parallel. Resultingly, it's better to use cluster mode for complex queries and local for simple ones.

Double-click (or enter) to edit

References:

[1]"The Big Mac Economic Index", Kaggle.com, 2022. [Online]. Available: https://www.kaggle.com/yamqwe/the-big-mac-economic-index. [Accessed: 18- Jan- 2022].

[2]"Spark SQL and DataFrames - Spark 3.2.0 Documentation", Spark.apache.org, 2022. [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html. [Accessed: 18- Jan- 2022].

[3]"Apache Spark DS DF and RDD", https://docs.google.com/presentation/d/194AYzBioTdgcgcdzpZmBEjmQ6fe6OTOz/edit#slide=id.p3, 2020.

Saved successfully!