Exercise 17: Performing DataFrame Operations in Spark

Let's start using Spark to perform input/output and simple aggregation operations. We want to read a CSV file, as we did before, to perform some analysis on it:

1. First, let's use the following command on Jupyter notebook to create a Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Python Spark Session").getOrCreate()
```

2. Now, let's use the following command to read the data from the people.csv file:

```
df = spark.read.csv('people.csv', header=True)
```

3.As we said before, Spark evaluation is lazy, so if we want to show what values are inside the DataFrame, we need to call the action, as illustrated here. This is not necessary with pandas: printing the DataFrame would work directly.

```
#df
#print(df)
df.show()
```

```
+------+---+------+
|  name|age|height|
+------+---+------+
|  Adam| 23|  1.98|
| Fiona| 34|  1.67|
|  Mary| 26|  1.72|
|George| 21|  1.83|
+------+---+------+
```

Exercise 18: Accessing Data with Spark

After reading our DataFrame and showing its contents, we want to start manipulating the data so that we can do an analysis. We can access data using the same NumPy selection syntax, providing the column name as input. Note that the object that is returned is of type Column:

1. Let's select one column from the DataFrame that we ingested in the previous exercise

```
df['age'].Column['age']
```

```
Column<b'age[Column][age]'>
```

2. Using the same DataFrame again, use the select method to select the name column:

```
df.select(df['age'])
```

```
DataFrame[age: string]
```

3. Now, it changed from Column to DataFrame. Because of that, we can use the methods for DataFrames. Use the show method for showing the results from the select method for age:

```
df.select(df['age']).show()
```

```
+---+
|age|
+---+
| 23|
| 34|
| 26|
| 21|
+---+
```

4. Let's select more than one column. We can use the names of the columns to do this:

```
df.select(df['age'],df['name']).show()
```

```
+---+------+
|age|  name|
+---+------+
| 23|  Adam|
| 34| Fiona|
| 26|  Mary|
| 21|George|
+---+------+
```

Exercise 20: Writing Data Back

As we saw with pandas, after performing some operations and transformations, let's say that we want to write the results back to the local file system. This can be very useful when we finish an analysis and want to share the results with other teams, or we want to show our data and results using other tools:

1. We can use the write method directly from the DataFrame:

```
df.write.csv('results12.csv', header=True)
```

Read it to see if it works:

```
df = spark.read.csv('results12.csv', header=True)
df.show()
```

```
+------+---+------+
|  name|age|height|
+------+---+------+
|  Adam| 23|  1.98|
| Fiona| 34|  1.67|
|  Mary| 26|  1.72|
|George| 21|  1.83|
+------+---+------+
```

Write to another format - JSON:

```
df.write.json('json_results.json')
```

Activity: Data Conversion

Convert data from cars.data file, used on previous lesson, from CSV format to spark JSON format. You can use pandas in this process. Notice differences between pandas JSON and spark JSON files.

Exercise 21: Writing Parquet Files

Let’s say that we received lots of CSV files and we need to do some analysis on them. We also need to reduce the data volume size. We can do that using Spark and Parquet. Use carsPARK.json file from previous activity.

1. Read the data

```
dfs = spark.read.json('carsSPARK.json')
```

```
dfs.show()
```

```
+---+------------+---------+------------+----------+----+--------------------+------+------+----+
|_c0|acceleration|cylinders|displacement|horsepower| mpg|                name|origin|weight|year|
+---+------------+---------+------------+----------+----+--------------------+------+------+----+
|  0|        12.0|        8|       307.0|     130.0|18.0|chevrolet chevell...|     1|3504.0|  70|
|  1|        11.5|        8|       350.0|     165.0|15.0|   buick skylark 320|     1|3693.0|  70|
|  2|        11.0|        8|       318.0|     150.0|18.0|  plymouth satellite|     1|3436.0|  70|
```

3. Now read the Parquet file to a new DataFrame (to see if it works):

```
dfs1=spark.read.parquet("dataFile")
```

```
dfs1.show()
```

```
+---+------------+---------+------------+----------+----+--------------------+------+------+----+
|_c0|acceleration|cylinders|displacement|horsepower| mpg|                name|origin|weight|year|
+---+------------+---------+------------+----------+----+--------------------+------+------+----+
|  0|        12.0|        8|       307.0|     130.0|18.0|chevrolet chevell...|     1|3504.0|  70|
|  1|        11.5|        8|       350.0|     165.0|15.0|   buick skylark 320|     1|3693.0|  70|
|  2|        11.0|        8|       318.0|     150.0|18.0|  plymouth satellite|     1|3436.0|  70|
|  3|        12.0|        8|       304.0|     150.0|16.0|       amc rebel sst|     1|3433.0|  70|
|  4|        10.5|        8|       302.0|     140.0|17.0|         ford torino|     1|3449.0|  70|
|  5|        10.0|        8|       429.0|     198.0|15.0|    ford galaxie 500|     1|4341.0|  70|
```

The write.parquet method creates a folder named data_file with a file with a long name such as part-00000-1932c1b2-e776-48c8-9c96-2875bf76769b-c000.snappy.parquet .

Exercise 22: Creating a Partitioned Dataset

Let’s create a partitioned dataset saved in Parquet from our cars database:

1. Read the data

```
dfs1=spark.read.parquet("dataFile")
```

2. Convert it to partitioned parquet

```
dfs1.write.parquet('dataFilePartitioned1',partitionBy=["year","cylinders"], compression="snappy")
```

```
dfs2=spark.read.parquet("dataFilePartitioned1")
```

Exercise 23: Parsing Text and Cleaning

In this exercise, we will read a text file, split it into lines and remove the words „**the”** and „**a”** from the string given string:

1.Read the text. we will use Resilient Distributed Dataset (RDD) API

```
df_rdd = spark.read.text("poem.txt").rdd
#dfs.show()
```

2. Extract the lines from the text using the following command:

```
lines = df_rdd.map(lambda line: line[0])
```

3. This splits each line in the file as an entry in the list.
To check the result, you can use the collect method, which gathers all data back to the driver process

```
lines.collect()
```

```
['To be, or not to be, that is the question:',
 "Whether 'tis nobler in the mind to suffer",
 'The slings and arrows of outrageous fortune,',
 'Or to take arms against a sea of troubles',
 'And by opposing end them. To die—to sleep,',
 'No more; and by a sleep to say we end',
 'The heart-ache and the thousand natural shocks',
 "That flesh is heir to: 'tis a consummation",
 "Devoutly to be wish'd. To die, to sleep;",
 "To sleep, perchance to dream—ay, there's the rub:",
 'For in that sleep of death what dreams may come,',
 'When we have shuffled off this mortal coil,',
 "Must give us pause—there's the respect",
 'That makes calamity of so long life.']
```

4. Now, let's count the number of lines, using the count method:

```
lines.count()
```

```
14
```

Be careful when using the **collect** method! If the DataFrame or RDD being collected is larger than the memory of the local driver, Spark will throw an error.

5. Now, let's first split each line into words, breaking it by the space around it, and combining all elements, removing words in uppercase:

```
#splits = lines.map(lambda x: x.split(' '))
splits = lines.flatMap(lambda x: x.split(' '))
```

```
#splits.collect()
```

```
lower_splits=splits.map(lambda x: x.lower().strip())
```

```
#lower_splits.collect()
```

6. Let's also remove the words the and a, and punctuations like '.', ',' from the given string.
First - prepare "dictionary".

```
prep = ['the', 'a', ',','.']
```

7. Use the following command to remove the stop words from our token list

```
tokens = lower_splits.filter(lambda x: x and x not in prep)
#tokens.collect()
```

We can now process our token list and count the unique words. The idea is to generate a list of tuples, where the first element is the token and the second element is the count of that particular token.

8. Let's map our token to a list:

```
token_list = tokens.map(lambda x: [x, 1])
```

9. Use the **reduceByKey** operation, which will apply the operation to each of the lists:

```
from operator import add
#countWords = token_list.reduceByKey(lambda a,b:a+b)
countWords = token_list.reduceByKey(add).sortBy(lambda x: x[1],ascending=False)
countWords.collect()
```

```
[('to', 11),
 ('that', 4),
 ('and', 4),
 ('of', 4),
 ('be,', 2),
 ('or', 2),
 ('is', 2),
 ("'tis", 2),
 ('in', 2),
 ('by', 2),
 ('end', 2),
 ('sleep,', 2),
 ('sleep', 2),
 ('we', 2),
 ('not', 1),
 ('question:', 1),
 ('whether', 1),
 ('nobler', 1),
 ('mind', 1),
 ('suffer', 1),
 ('slings', 1),
```