E10-1 (2) (2)

October 31, 2023

Name : Kattirsitti Jeet Govindrao

Roll no.: 22B0010

0.1 E10-1

This Notebook illustrates the use of "MAP-REDUCE" to calculate averages from the data contained in nsedata.csv.

0.1.1 Task 1

You are required to review the code (refer to the SPARK document where necessary), and add comments / markup explaining the code in each cell. Also explain the role of each cell in the overall context of the solution to the problem (ie. what is the cell trying to achieve in the overall scheme of things). You may create additional code in each cell to generate any debug output that you may need to complete this exercise. ### Task 2 You are required to write code to solve the problem stated at the end this Notebook ### Submission Create and upload a PDF of this Notebook. BEFORE CONVERTING TO PDF and UPLOADING ENSURE THAT YOU REMOVE / TRIM LENGTHY DEBUG OUTPUTS . Short debug outputs of up to 5 lines are acceptable.

[91]:
```
!pip install findspark
!pip install pyspark
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: findspark in
/home/hduser/.local/lib/python3.10/site-packages (2.0.1)
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pyspark in
/home/hduser/.local/lib/python3.10/site-packages (3.5.0)
Requirement already satisfied: py4j==0.10.9.7 in
/home/hduser/.local/lib/python3.10/site-packages (from pyspark) (0.10.9.7)
```

[92]:
```
import findspark
findspark.init()
# In PySpark, findspark.init() is a function used to initialize the findspark␣
 ↪library, which helps to locate and set up the Spark environment on the VM .
# using findspark.init() SparkContext and SparkSession can be created.
```

[93]:
```
import pyspark
from pyspark.sql.types import *
# Importing PySpark - the Python API for Apache Spark.
# All the functions are imported from pyspark.sql.types which are used to work␣
 ↪with a dataset having different datatype.
# Can also be written as import pyspark.sql.types instead.
```

Creating a SparkContext in PySpark with the application name "E10."

[ ]:
```
sc = pyspark.SparkContext(appName="E10_1")
```

[95]:
```
sc
```

[95]:
```
<SparkContext master=local[*] appName=Task2_final>
```

[96]:
```
rdd1 = sc.textFile("/home/hduser/spark/nsedata.csv")
# Loading of the content of file into an RDD framework.
```

[97]:
```
rdd1 = rdd1.filter(lambda x: "SYMBOL" not in x)
# .filter() is an RDD transformation operation in Apache Spark. It takes a␣
 ↪function as an argument and returns a new RDD containing only the elements␣
 ↪that satisfy the condition defined in the function.
#  Above implementation results in the exclusion any lines that contain the␣
 ↪word "SYMBOL removal of headers.
```

[98]:
```
rdd2 = rdd1.map(lambda x : x.split(","))
# Just like a map phase in Hadoop's mapreduce framework, but it's not␣
 ↪generating key-value pairs as in the traditional Hadoop MapReduce framework.␣
 ↪Instead, it's splitting each line in rdd1
```

Map transformations to map opening price(X[2]) of each Symbol(X[0],key)

Map transformations to map closing price(X[5]) of each Symbol(X[0],key)

[99]:
```
rdd_open = rdd2.map(lambda x : (x[0]+"_open",float(x[2])))
rdd_close = rdd2.map(lambda x : (x[0]+"_close",float(x[5])))
```

[100]:
```
rdd_united = rdd_open.union(rdd_close)
# Union transformation applied for obtaining both the RDD'S stacked.
```

[101]:
```
reducedByKey = rdd_united.reduceByKey(lambda x,y: x+y)
# Since there are multiple occurences of symbol becuse of unioning␣
 ↪transformation hence value pairs are being added for the common key.
```

[102]:
```
temp1 = rdd_united.map(lambda x: (x[0],1)).countByKey()
countOfEachSymbol = sc.parallelize(temp1.items())
# First command maps the RDD to value  pair 1 and then the .countByKey() method␣
 ↪is used for counting the total of the value which ends up being the total␣
 ↪count of a distinct symbol across RDD's.
```

[103]:
```
symbol_sum_count = reducedByKey.join(countOfEachSymbol)
# Join is different form union since join is used only when the 2 RDD's share a common key.
```

[104]:
```
averages = symbol_sum_count.map(lambda x : (x[0], x[1][0]/x[1][1]))
# x[0] has symbol name and x[1]x[0] has the sum of all the opening and closing prices,x[1][1] has the sum of all the opening and closing occurences of that particular symbol
```

[105]:
```
averagesSorted = averages.sortByKey()
# .sortByKey() will sort them in ascending lexicographical (alphabetical) order.
```

[106]:
```
averagesSorted.saveAsTextFile("/home/hduser/spark/averages2")
# Saved in the directory "/home/hduser/spark/averages" with the filename averages.Sorted.
```

[107]:
```
sc.stop()
# Spark session is stopped.
```

0.1.2 Review the output files generated in the above step and copy the first 15 lines of any one of the output files into the cell below for reference. Write your comments on the generated output

11 output files were generated containing average open, close price data. Generated output files denote that averages were computed across different RDDs.
('20MICRONS_close', 53.004122877930484) ('20MICRONS_open', 53.32489894907032) ('3IINFOTECH_close', 18.038803556992725) ('3IINFOTECH_open', 18.17417138237672) ('3MINDIA_close', 4520.343977364591) ('3MINDIA_open', 4531.084518997574) ('3RDROCK_close', 173.2137755102041) ('3RDROCK_open', 173.18316326530612) ('8KMILES_close', 480.73622047244095) ('8KMILES_open', 481.63858267716535) ('A2ZINFRA_close', 18.609433962264156) ('A2ZINFRA_open', 18.73553459119497) ('A2ZMES_close', 89.69389505549951) ('A2ZMES_open', 90.46271442986883) ('AANJANEYA_close', 441.84030249110316)

0.2 Task 2 - Problem Statement

0.2.1 Using the MAP-REDUCE strategy, write SPARK code that will create the average of HIGH prices for all the traded companies, but only for any 3 months of your choice. Create the appropriate (K,V) pairs so that the averages are simultaneously calculated, as in the above example. Create the output files such that the final data is sorted in descending order of the company names.

Order of Header in the CSV file "symbol","series","open","high","low","close","last","prevclose","tottrdqty","tottr

[108]:
```
import findspark
findspark.init()

import pyspark
from pyspark.sql.types import *
```

[109]:
```
sc=pyspark.SparkContext(appName="Task2_f")
```

[110]:
```
sc
```

[110]:
```
<SparkContext master=local[*] appName=Task2_f>
```

[111]:
```
rdd1 = sc.textFile('/home/hduser/spark/nsedata.csv')
rdd1 = rdd1.filter(lambda x:"SYMBOL" not in x)
# RDD created with only required data and no headers.
```

[112]:
```
rdd2 = rdd1.map(lambda x: x.split(","))
# Allows us to access field values easily by using indexing just like in arrays.
 ↪
```

[113]:
```
rdd_month = rdd2.map(lambda x: (x[10].split('-')[1] + "_Month",float(x[3])))
```

[114]:
```
rdd_reduce = rdd_month.reduceByKey(lambda x,y:x+y)
```

[115]:
```
rdd_count = rdd_month.map(lambda x:(x[0],1)).countByKey()
count = sc.parallelize(rdd_count.items())
```

[116]:
```
rdd_avg = rdd_reduce.join(count)
```

[117]:
```
Avg = rdd_avg.map(lambda x:(x[0],x[1][0]/x[1][1]))
Avg_sort = Avg.sortByKey(ascending=False)
```

[118]:
```
Avg_sort.take(3)
```

[118]:
```
[('SEP_Month', 360.1827821256352),
 ('OCT_Month', 359.693761531464),
 ('NOV_Month', 369.2610847251277)]
```

[119]:
```
Avg_sort.saveAsTextFile('/home/hduser/spark/Task2_end')
```

Some of the chosen months ('FEB_Month', 364.83198766865087) ('DEC_Month', 362.56428352680183) ('SEP_Month', 360.1827821256352)

[120]:
```
sc.stop
```

[120]:
```
<bound method SparkContext.stop of <SparkContext master=local[*]
appName=Task2_f>>
```