ECS765P - Big Data Processing

Coursework – ETHEREUM ANALYSIS

Akansh Katyayan
210282553

PART A: TIME ANALYSIS (20%)

Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset. Create a bar plot showing the average value of transactions in each month between the start and end of the dataset. Note: As the dataset spans multiple years and you are aggregating together all transactions in the same month, make sure to include the year in your analysis.
Note: Once the raw results have been processed within Hadoop/Spark you may create your bar plot in any software of your choice (excel, python, R, etc.)

Task 1: Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.
MapReduce program using mrjob library has been used as below:

Python Code: parta_task1.py

PART_A > Task1 > parta_task1.py

```python
from mrjob.job import MRJob
import time


class PartA(MRJob):
    '''
    Class to get number of Ethereum transactions occurring every month each year.
    '''
    def mapper(self, _, line):
        '''
        mapper to split records and check timestamp of each transaction.
        Create Key as month and year and value as 1 for each transaction
        '''
        try:
            fields = line.split(",")
            if (len(fields) == 7):
                epoch_time=int(fields[6]) # Timestamp value
                date_year = time.strftime("%Y", time.gmtime(epoch_time))
                date_month = time.strftime("%m", time.gmtime(epoch_time))
                yield((date_year, date_month), 1)
        except:
            pass

    def combiner(self, key, value):
        yield(key, sum(value))

    def reducer(self, key, value):
        '''
        Reducer sums all the individual transactions occurring on same month and year
        '''
        yield(key, sum(value))


if __name__ == '__main__':
    PartA.run()

```

Execution Command:
python parta_task1.py -r hadoop --output-dir coursework/PartATask1 --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions

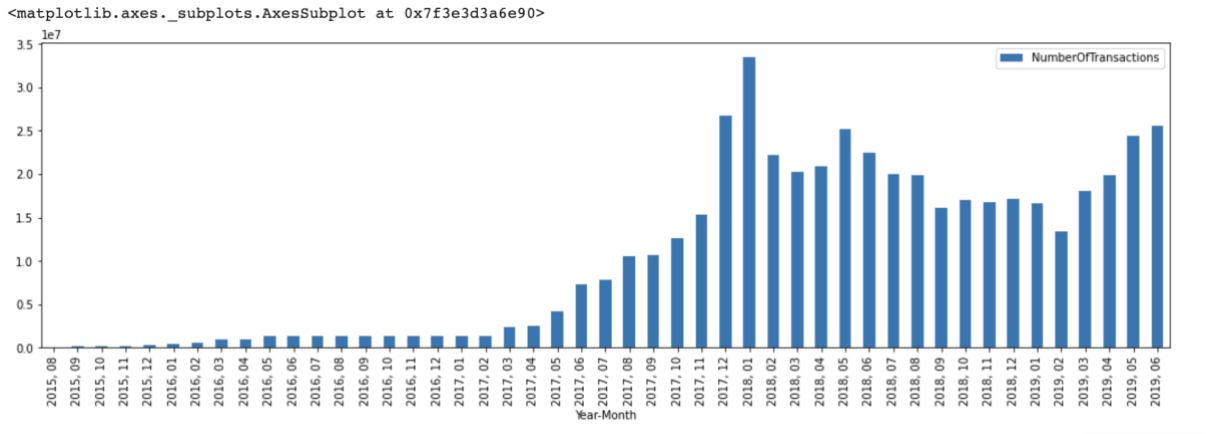
Execution Job ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1186/

Output File: OutPartATask1.csv

```
["2015", "09"]   173805
["2015", "10"]   205045
["2015", "12"]   347092
["2016", "02"]   520040
["2016", "04"]   1023096
["2016", "06"]   1351536
["2016", "08"]   1405743
["2016", "11"]   1301586
["2017", "01"]   1409664
["2017", "03"]   2426471
["2017", "05"]   4245516
["2017", "07"]   7835875
["2017", "09"]   10672734
["2017", "10"]   12570063
["2017", "12"]   26687692
["2018", "02"]   22231978
["2018", "04"]   20876642
["2018", "06"]   22471788
["2018", "08"]   19842059
["2018", "11"]   16713911
["2019", "01"]   16569597
["2019", "03"]   18029582
["2019", "05"]   24332475
["2015", "08"]   85609
["2015", "11"]   234733
["2016", "01"]   404816
["2016", "03"]   917170
["2016", "05"]   1346796
["2016", "07"]   1356907
["2016", "09"]   1387412
["2016", "10"]   1329847
["2016", "12"]   1316131
["2017", "02"]   1410048
["2017", "04"]   2539966
["2017", "06"]   7244657
["2017", "08"]   10523178
["2017", "11"]   15292269
["2018", "01"]   33504270
["2018", "03"]   20261862
["2018", "05"]   25105717
["2018", "07"]   19937033
["2018", "09"]   16056742
["2018", "10"]   17056926
["2018", "12"]   17107601
["2019", "02"]   13413899
["2019", "04"]   19830158
["2019", "06"]   25613628
```

Bar Plot: Bar plot has been plotted using Python Jupyter notebook. Attached with the code.



Explanation: Transaction dataset contains each transaction with value in Wei with its timestamp (time of the transaction). Timestamp data is used to fetch month and year of each transaction by using the time library in Python. In a mapper job, combination of month and year is considered as a key to avoid the duplication by month value and value '1' is yielded for each record.
The combiner is used to aggregate the transactions occurred in the same month and year (i.e., having the same key) by summing up the number of transactions.

Finally, the reducer performs the same function as combiner which is used to increase the overall performance of the job. Reducer yields all the distinct key values (unique month and year) with total number of transactions occurred in each month and year.

Output data generated in OutPartATask1.csv is used to plot a graph using Pandas and Matplotlib where data was sorted by month and year and number of transactions represents the height of each bar in the above plot.

Task 2: Create a bar plot showing the average value of transactions in each month between the start and end of the dataset.

MapReduce is used to calculate average value of transactions in each month.

Python Code: parta_task2.py

PART_A > Task2 > parta_task2.py

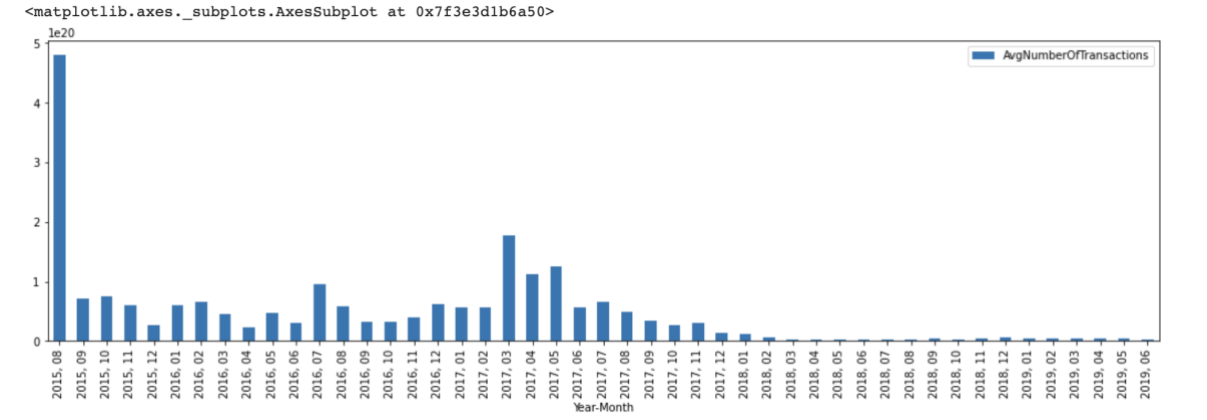
```python
from mrjob.job import MRJob
import time


class PartATask2(MRJob):
    """
    Average value of transactions in each month between the start and end of the dataset.
    """
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            value = int(fields[3])
            epoch_time = float(fields[6]) #Timestamp
            date_year = time.strftime("%Y", time.gmtime(epoch_time))
            date_month = time.strftime("%m", time.gmtime(epoch_time))
            yield((date_year, date_month), (1, value))
        except:
            pass

    def combiner(self, key, value):
        count = 0
        total_value = 0
        for val in value:
            count += val[0]
            total_value += val[1]
        yield(key, (count, total_value))

    def reducer(self, key, value):
        count = 0
        total_value = 0
        for val in value:
            count += val[0]
            total_value += val[1]
        yield(key, total_value/count)

if __name__ == '__main__':
    PartATask2.run()

```

Execution Command:
python parta_task2.py -r hadoop --output-dir coursework/PartATask2 --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions
Execution Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1202/

Output File: OutPartATask2.csv

```
["2015", "09"]   7.0464679457674535e+19
["2015", "10"]   7.416931809333862e+19
["2015", "12"]   2.6764096183940583e+19
["2016", "02"]   6.55476087599054e+19
["2016", "04"]   2.2670632047054135e+19
["2016", "06"]   3.0490334850065605e+19
["2016", "08"]   5.908198737290209e+19
["2016", "11"]   3.964431643835122e+19
["2017", "01"]   5.620285956535166e+19
["2017", "03"]   1.770215809422227e+20
["2017", "05"]   1.2484777365193273e+20
["2017", "07"]   6.4981463792721e+19
["2017", "09"]   3.438888521827938e+19
["2017", "10"]   2.6736910424748003e+19
["2017", "12"]   1.3737046580115218e+19
["2018", "02"]   6.230362795090817e+18
["2018", "04"]   2.449268234852099e+18
["2018", "06"]   2.8085234163056374e+18
["2018", "08"]   2.3989507981126523e+18
["2018", "11"]   5.397909048901716e+18
["2019", "01"]   4.45481388902492e+18
["2019", "03"]   3.823274668677927e+18
["2019", "05"]   4.005277417445827e+18
["2015", "08"]   4.8052118459597835e+20
["2015", "11"]   5.948474386250283e+19
["2016", "01"]   6.106607047719591e+19
["2016", "03"]   4.5853064127780815e+19
["2016", "05"]   4.7046609524468195e+19
["2016", "07"]   9.577823510582716e+19
["2016", "09"]   3.2627612247557157e+19
["2016", "10"]   3.2444426339709395e+19
["2016", "12"]   6.146658677538069e+19
["2017", "02"]   5.558009016262998e+19
["2017", "04"]   1.1135007462190998e+20
["2017", "06"]   5.678772230936606e+19
["2017", "08"]   4.827395651885326e+19
["2017", "11"]   2.96411032747405e+19
["2018", "01"]   1.11645727207502e+19
["2018", "03"]   2.728079891162356e+18
["2018", "05"]   2.4981909136531256e+18
["2018", "07"]   2.2749347554396106e+18
["2018", "09"]   3.733481101982448e+18
["2018", "10"]   3.070402143025912e+18
["2018", "12"]   5.944894163484742e+18
["2019", "02"]   3.980845264511403e+18
["2019", "04"]   4.1280245320091075e+18
["2019", "06"]   3.0362427767092367e+18
```

Bar Plot: Bar plot has been plotted using Python Jupyter notebook. Attached with the code.



Explanation:
Similar Mapper function has been used for task 2 to get each transaction. Here, it also includes value of each transaction from 'value' column of the 'transactions' dataset. Therefore, mapper yields 1 for each transaction and value for its value in Wei for each key (month and year).

The combiner is again used to improve the aggregation process and overall performance of the job. It counts the number of transactions for each unique month and year and aggregate the value of all the transactions in the same month and year.

Subsequently, the reducer also does the same function as combiner and yields the average value of transactions in each month and year by dividing the sum of the values of transactions in the given month and year by the total number of transactions in the same month and year.
Finally, Bar plot is plotted using the output data OutPartATask2.csv in python Jupyter notebook.

PART B: TOP TEN MOST POPULAR SERVICES (25%)

Evaluate the top 10 smart contracts by total Ether received. An outline of the subtasks required to extract this information is provided below, focusing on a MRJob based approach.
Job1: INITIAL AGGREGATION
Job2: JOINING TRANSACTIONS/CONTRACTS AND FILTERING
Job3: TOP TEN

There are different ways to handle this problem. Three different MapReduce jobs can be written where first job calculates the aggregated values of transactions for each to_address. Second job to identify smart contracts using replication join between transactions dataset and contracts dataset and finally the third job which identifies the top 10 smart contracts.

Instead of having three different job, same can be achieved in a single MapReduce job with 2 different mappers and reducers processed one after the other using MRStep library in Python. Below is the code for the same.

Python Code: partb_alltasks.py

PART_B > partb_alltasks.py

```python
from mrjob.job import MRJob
from mrjob.step import MRStep

class PARTB(MRJob):
    """
    Combined Class to Implement
    Job 1 (Initiial Aggregation),
    Job 2 (Join Transactions and Contracts),
    Job 3 (Top Ten)

    """
    def mapper1(self, _, line):
        fields = line.split(',')
        try:
            #Check if number of fields are 7 => Transaction Dataset
            if len(fields) == 7:
                to_address = fields[2]
                value = int(fields[3])
                # 1 here is and identifier
                yield(to_address, ('Transactions', value))

            # check if number of fields are 5 -> Contract Dataset
            elif len(fields) == 5:
                address1 = fields[0]
                #2 here is and identifier
                yield(address1, ('Contracts', None))
        except:
            pass

```

```python
    def reducer1(self, key, values):
        tempvar = False
        allvalues = []
        for val in values:
            #if in Transaction Dataset
            if val[0]=='Transactions':
                allvalues.append(val[1])
            #if in Contract Dataset
            elif val[0] == 'Contracts':
                tempvar = True
        if tempvar:
            # if tempvar is True, it means the address is present in both
            # transaction and contract Dataset, hence is popular
            yield(key, sum(allvalues))

    def mapper2(self, key, value):
        yield(None, (key, value))

    def reducer2(self, _, value):
        sortedvalues = sorted(value, reverse = True, key = lambda x: x[1])
        for val in sortedvalues[:10]:
            yield(val[0], val[1])

    def steps(self):
        return [MRStep(mapper = self.mapper1, reducer=self.reducer1), MRStep(mapper = self.mapper2, reducer = self.reducer2)]

if __name__ == '__main__':
    PARTB.run()

```

Execution command:
python partb_alltasks.py -r hadoop --output-dir coursework/PartBAllTasks --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts

Execution Job Id:
Two job ids are generated as the program runs two reducer jobs step by step:
Job ID 1:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1226/

Job ID 2:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1239/

Output File: OutPartB.txt

```
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	841551008099658658227267776
"0xfa52274dd61e1643d2205169732f29114bc240b3"	45787484483189352986478805
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"	45620624001350712557268573
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"	43170356092262468919298969
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"	27068921582019542499882877
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"	21104195138093660050000000
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"	15562398956802112254719409
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"	11983608729202893846818681
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"	11706457177940895521770404
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"	83790007519177556240575000
```

Explanation:
Transactions and Contracts datasets are read together by the job and used based on the number of columns. First mapper checks, If the number of columns is 7, to_address and value of transaction are read and in case of 5 columns, only the address is read from the contracts data. For a transaction to be a popular service, the to_address of the transactions data should be available as an address in the Contracts dataset.

Therefore, Reducer 1 aggregates the values based on the to_address from transactions dataset and yields the address and aggregated value only if the address is available in the Contracts dataset.

Mapper 1 and Reducer 1, thus solves the requirement of Job 1 and Job 2 tasks of Part B.

Further, Mapper 2 takes the output of reducer 1 and yields the same data as there is no further processing required. Finally, Reducer 2 sorts and yields the top 10 values from the popular services.

Output contains the top 10 Addresses of the transactions and aggregated value of these transactions.

PART C: TOP TEN MOST ACTIVE MINERS (15%)

Evaluate the top 10 miners by the size of the blocks mined. This is simpler as it does not require a join. You will first have to aggregate blocks to see how much each miner has been involved in. You will want to aggregate size for addresses in the miner field. This will be similar to the wordcount that we saw in Lab 1 and Lab 2. You can add each value from the reducer to a list and then sort the list to obtain the most active miners.

Given task to evaluate top 10 miners is implemented using Map Reduce.

Python Code: partc.py

```python
from mrjob.job import MRJob
from mrjob.step import MRStep


class PartC(MRJob):
    '''
    Class to evaluate top 10 miners by the size of the block mined.
    Blocks dataset is used.
    '''
    def mapper1(self, _, line):
        '''
        mapper to split records and check timestamp of each transaction.
        Create Key as month and year and value as 1 for each transaction
        '''
        try:
            fields = line.split(",")
            if (len(fields) == 9):
                miner_block = fields[2]
                size = int(fields[4])
                yield(miner_block, size)

        except:
            pass

    def reducer1(self, key, value):
        '''
        Reducer sums all the individual size values of each distinct miner
        '''
        yield(key, sum(value))

    def mapper2(self, key, size_values):
        """
        Mapper 2 to pass key and values as the new values as no further processing required
        """
        try:
            yield(None, (key, size_values))
        except:
            pass

    def reducer2(self, _, size_value):
        """
        Reducer 2 to sort size value on decreasing order and yield top 10 miners based on size
        """
        try:
            sorted_size_values = sorted(size_value, reverse=True, key=lambda x: x[1])
            for miner_size in sorted_size_values[:10]:
                yield(miner_size[0], miner_size[1])

        except:
            pass

    def steps(self):
        return [MRStep(mapper = self.mapper1, reducer=self.reducer1), MRStep(mapper = self.mapper2, reducer = self.reducer2)]

if __name__ == '__main__':
    PartC.run()

```

Execution Command:
python partc.py -r hadoop --output-dir coursework/OutPartC --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/blocks

Execution Job Id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_1326/

Output: OutPartC.txt

```
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"    23989401188
"0x829bd824b016326a401d083b33d092293333a830"    15010222714
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"    13978859941
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"    10998145387
"0xb2930b35844a230f00e51431acae96fe543a0347"    7842595276
"0x2a65aca4d5fc5b5c859090a6c34d164135398226"    3628875680
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"    1221833144
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"    1152472379
"0x1e9939daaad6924ad004c2560e90804164900341"    1080301927
"0x61c808d82a3ac53231750dadc13c777b59310bd9"    692942577
```

Explanation:
In the given problem, MRStep library of python is used to execute 2 mappers and reducers, step by step to perform the required operations.
Mapper 1 splits the block dataset into columns and yields Miner and corresponding size of the ETH. Reducer 1 is used to aggregate the size of each distinct miner block by using sum(values).

Mapper 2 reads the output yielded by reducer 1 and yields the same in the form of value without any key as no further processing is required in mapper 2.

Finally, Reducer 2 sorts the values in the decreasing order based on aggregated size values and yields the top 10 Miners.

PART D: Data exploration (40+%)

Task 1: SCAM ANALYSIS

1. ***Popular Scams:*** *Utilising the provided scam dataset, what is the most lucrative form of scam? Does this correlate with certainly known scams going offline/inactive? For the correlation, you could produce the count of how many scams for each category are active/inactive/offline/online/etc and try to correlate it with volume in gas to make conclusions on whether state plays a factor in making some scams more lucrative. Therefore, getting the volume and state of each scam, you can make a conclusion whether the most lucrative ones are ones that are online or offline or active or inactive. So for that purpose, you need to just produce a table with SCAM TYPE, STATE, VOLUME which would be enough (15%).*

Implemented using Spark

Python Code: Scam_analysis.py

Scam_analysis.py

PART_D > ScamAnalysis > Scam_analysis.py

```python
from pyspark.sql.functions import *
import pyspark
import json

def is_good_line(line):
    try:
        fields = line.split(',')
        if len(fields) == 7: # transactions
            str(fields[2]) # to_address
            if int(fields[3]) == 0: # skip data with value = 0
                return False
            return True
    except:
        return False

sc = pyspark.SparkContext()
# print Spark Objecct Application ID
print(sc.applicationId)

# Context for CSV Data
newcontext = pyspark.SQLContext(sc)

# Read CSV using SQL Context object
df_scams = newcontext.read.option('header', False).format('csv').load("scams.csv")

# Convert SQL Context object into RDD
scams_rdd = df_scams.rdd.map(lambda x: (x[0],(x[1],x[2])))

# Read Transactions dataset
transactions = sc.textFile('/data/ethereum/transactions').filter(is_good_line)

"""fetch required fields from transactions dataset
[2]: to_address
[3]: values
[4]: Gas
"""
step1 = transactions.map(lambda l:  (l.split(',')[2], (float(l.split(',')[4]), float(l.split(',')[3]))))

# Join Transactions dataset with Scams records
step2 = step1.join(scams_rdd)

# Sum values and gas values based on Scam Type
step_add = step2.map(lambda x: (x[1][1][0], (x[1][0][0], x[1][0][1])))

# Reduce by Scam Type as a key
sum_reduce= step_add.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1]))
sum_reduce.saveAsTextFile('output_scams_analysis_part1_new')

# Part 2
# Key as Scam Type + Status , Value as Gas
new_map = step2.map(lambda x: (x[1][1],  (x[1][0][0], x[1][0][1])))

# Reduce by key - Scam Type + Status : Value as Gas
new_value = new_map.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1])).sortByKey(ascending=True)
new_value.saveAsTextFile('output_scams_analysis_part2_new')
```

Execution Command: Given problem is handled using below steps:

1. Download Scams.json file to local
   Command: *hadoop fs -copyToLocal /data/Ethereum/scams.json .*
2. Executed python code to convert JSON to CSV
   Command: *python3 convert_json_to_csv.py*
3. Copy scams.csv to Hadoop
   Command: *hadoop fs -copyFromLocal scams.csv .*
4. Execute Spark code:
   Command: *spark-submit Scams_analysis.py*

Execution Job Id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_5958

Output:
Part 1: output_scamanalysis_part1.txt

```
(u'Phishing', (724922480.0, 4.372701002504075e+22))
(u'Fake ICO', (7402302.0, 1.35645756688963e+21))
(u'Scamming', (5978937963.0, 4.47158060984405e+22))
```

Part 2: output_scamanalysis_part2.txt

```
((u'Fake ICO', u'Offline'), (7402302.0, 1.35645756688963e+21))
((u'Phishing', u'Active'), (111750772.0, 6.256455846464806e+21))
((u'Phishing', u'Inactive'), (2255249.0, 1.488677770799503e+19))
((u'Phishing', u'Offline'), (610020459.0, 3.745402749273795e+22))
((u'Phishing', u'Suspended'), (896000.0, 1.63990813e+18))
((u'Scamming', u'Active'), (4282186989.0, 2.2612205279194257e+22))
((u'Scamming', u'Offline'), (1694248234.0, 2.2099890651296327e+22))
((u'Scamming', u'Suspended'), (2502740.0, 3.71016795e+18))
```

Explanation:
Below steps have been implemented for the same.

1. Downloaded scams.json data from Hadoop cluster to local machine
2. Used python code: convert_json_to_csv.py to convert it into a csv file scams.csv by splitting each address from each scam as a separate row and extracted scan type and scam status values corresponding to each address.
3. Uploaded scams.csv to Hadoop cluster
4. Spark code to get the results

In the first part, the task is to find the most lucrative type of scam.
First, SQLContext is used to create dataframe which created using the Scams.csv. The dataframe is then converted into a Spark RDD. Transactions dataset from Hadoop cluster is stored in another RDD by reading it using spark context. From the 'transactions' dataset, each address is used as a key and values and gas is used as a value against each address. To join the transactions data with scams data, join operation is used on address as a key. To compute gas and values for each scam type, a new RDD is created by mapping gas and values against each Scam type. Further, all the gas

and values are added based on scam type by using ReduceByKey function of PySpark. Output implies that **Scamming is the most lucrative type of scam.**

In the second part, the task is to find the correlation between scam type and scam status.
To compute the correlation, volume of 'gas' and 'value' both are used for each scam type under each status has been calculated by using a combination of Scam type and Scam status as a key in the mapping function and fetching 'gas' and 'value' value against this key. Further, all the 'gas' and 'values' value for each key are added respectively by using reduceByKey method with a key (Scam Type, Scam Status) and value as 'Gas' and 'Value'.

Conclusion

The output of the correlation implies that the Gas value is more when the scam is active or in an offline status. Gas value reduces when the scam is Inactive and is the minimum when the scam is Suspended.

Task 2: Gas Guzzlers

For any transaction on Ethereum a user must supply gas. How has gas price changed over time? Have contracts become more complicated, requiring more gas, or less so? Also, could you correlate the complexity for some of the top-10 contracts found in Part-B by observing the change over their transactions

Task 2 - Part 1: *How has gas price changed over time?*

Map Reduce is used to solve the given problem.

Python Code: gg_part1.py

PART_D > GasGuzzlers > Part1 > gg_part1.py

```python
from mrjob.job import MRJob
import time


class GasGuzzlerPart1(MRJob):
    """
    Function to check the gas price change with time
    """
    def mapper(self, _, lines):
        try:
            fields = lines.split(",")
            if len(fields) == 7:
                gas_price = int(fields[5]) # Gas Price from Transactions Dataset
                date = time.gmtime(float(fields[6]))
                key = str(date.tm_year) + '-' + str(date.tm_mon) # Year and Month as Key
                yield(key, gas_price)
        except:
            pass

    def reducer(self, key, values):
        sum_price = 0
        count = 0

        for value in values:
            sum_price = sum_price + value
            count = count + 1
        # Average of Gas Price per Year-Month
        yield(key, float(sum_price/count))


if __name__ == '__main__':
    GasGuzzlerPart1.run()

```

Execution Command:
python gg_part1.py -r hadoop --output-dir coursework/PartD_GGPart1 --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions

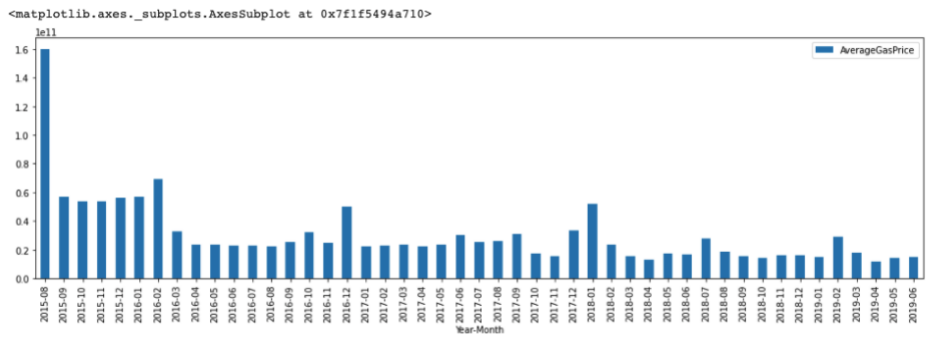
Execution Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_6152/

Output: gg_part1.csv

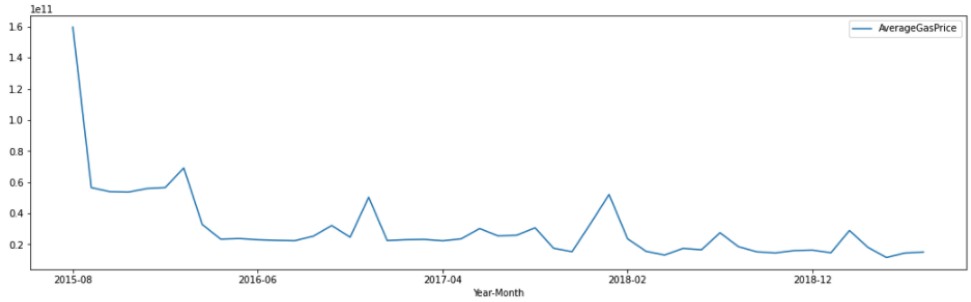
PART_D > GasGuzzlers > Part1 > gg_part1.csv

```
"2015-11"	53607614201.796776
"2015-8"	159744029578.03113
"2016-1"	56596270931.31685
"2016-10"	32112869584.914665
"2016-12"	50318068074.68128
"2016-3"	32797039087.356667
"2016-5"	23746277028.263245
"2016-7"	22629542449.24175
"2016-9"	25270403393.626083
"2017-11"	15312465314.693544
"2017-2"	23047230327.254303
"2017-4"	22355124545.395317
"2017-6"	30199442465.128727
"2017-8"	25905774673.990257
"2018-1"	52106060636.84502
"2018-10"	14526936383.350008
"2018-12"	16338844844.014668
"2018-3"	15549765961.743273
"2018-5"	17422505108.986416
"2018-7"	27506077453.154327
"2018-9"	15213870989.523378
"2019-2"	28940599438.14876
"2019-4"	11569797163.994152
"2019-6"	15076119023.332266
"2015-10"	53901692120.53661
"2015-12"	55899526672.35486
"2015-9"	56511301521.033226
"2016-11"	24634294365.279953
"2016-2"	69180681134.38849
"2016-4"	23361180502.721268
"2016-6"	23021251389.812134
"2016-8"	22396836435.95849
"2017-1"	22507570807.719795
"2017-10"	17506742628.97231
"2017-12"	33423472930.410748
"2017-3"	23232253600.81683
"2017-5"	23572314972.01526
"2017-7"	25460300456.232986
"2017-9"	30675106219.637596
"2018-11"	16034859008.681648
"2018-2"	23636574203.828976
"2018-4"	13153739247.92998
"2018-6"	16533308366.813036
"2018-8"	18483235826.894573
"2019-1"	14611816445.785303
"2019-3"	18083035519.522877
"2019-5"	14479858767.711182
```

Bar Plot: Bar plot has been plotted using Python Jupyter notebook. Attached with the code.



```
<matplotlib.axes._subplots.AxesSubplot at 0x7f1f547de690>
```



Explanation:
To compute the gas price change with time, transactions dataset has been used to fetch value of gas price and timestamp. Month and Year values are fetched from the timestamp value using the time library. Mapper yields the key as year-month and value as gas price.

Reducer is then used to add the value of gas prices for each key and counts the number of transactions to calculate the average gas prices per month-year.

Above Bar plot and Line plots show that the gas-price change is at the peak when started in Aug 2018 and then decreased. However, some sudden rise (peaks) can be observed in the gas price during Nov to Feb months of every year.

Task 2 - Part 2: *Have contracts become more complicated, requiring more gas, or less so?*

Implemented using Spark
Python Code: gg_part2.py

```python
import pyspark
import time

def check_good_contracts(line):
    """
    Function to check if the data is contracts types and contains values for each column
    """
    try:
        fields = line.split(',')
        if len(fields) != 5:
            return False
        float(fields[3]) # Block Number
        return True
    except:
        return False

def check_good_block(line):
    """
    Function to check if the data is Blocks datasets and contains value for each column
    It also sets the below columns as float values
    """
    try:
        fields = line.split(',')
        if len(fields)!=9:
            return False
        float(fields[0]) # Block Number
        float(fields[6]) # gas_used
        float(fields[7]) # timestamp
        return True
    except:
        return False

# Create Spark Context object
sc = pyspark.SparkContext()

# Print Spark Job Id
print(sc.applicationId)

# Read Contracts Data set from hadoop cluster
lines1 = sc.textFile('/data/ethereum/contracts')
cleaned_contracts = lines1.filter(check_good_contracts)
contracts_block_number = cleaned_contracts.map(lambda l: (l.split(',')[3], 1))

# Read Blocks Data set from hadoop cluster
lines2 = sc.textFile('/data/ethereum/blocks')
cleaned_blocks = lines2.filter(check_good_block)

```

```
"""
[0]: block number, [3]: difficulty, [6]: gas_used, [7]: timestamp
"""
mapper_block = cleaned_blocks.map(lambda l: (l.split(',')[0], (int(l.split(',')[3]),int(l.split(',')[6]), time.strftime("%Y-%m", time.gmtime(float(l.split(',')[7]))))))
print(mapper_block.take(2))
# [(u'4776199', (1765656009004680, 2042230, '2017-12')), (u'4776200', (1765656009037448, 4385719, '2017-12'))]

results = mapper_block.join(contracts_block_number)
print(results.take(2))
# [(u'7352442', ((1862016659491710, 6317667, '2019-03'), 1)), (u'7352442', ((1862016659491710, 6317667, '2019-03'), 1))]

updated_result = results.map(lambda x: (x[1][0][2], (x[1][0][0], x[1][0][1], x[1][1])))
print(updated_result.take(2))
# [('2017-08', (1600249491213932, 2032838, 1)), ('2019-06', (2134143517862686, 7989075, 1))]

final = updated_result.reduceByKey(lambda a,b: (a[0]+b[0], a[1]+b[1], a[2]+b[2]))
print(final.take(2))
# [('2019-02', (10840825281991818398629L, 3197663942850, 419164)), ('2016-07', (1662703037969682192, 43543836232, 28773))]

updated_final = final.map(lambda x: (x[0], (float(x[1][0] / x[1][2]), float(x[1][1] / x[1][2])))).sortByKey(ascending=True)
print(updated_final.take(2))
# [('2015-08', (4030960805570.0, 360218)), ('2015-09', (6577868584193, 540131))]

#Output is of the format Year-Month, Complexity, Avg Gas
updated_final.saveAsTextFile('output_gg_part2')
```

Execution Command:
spark-submit gg_part2.py

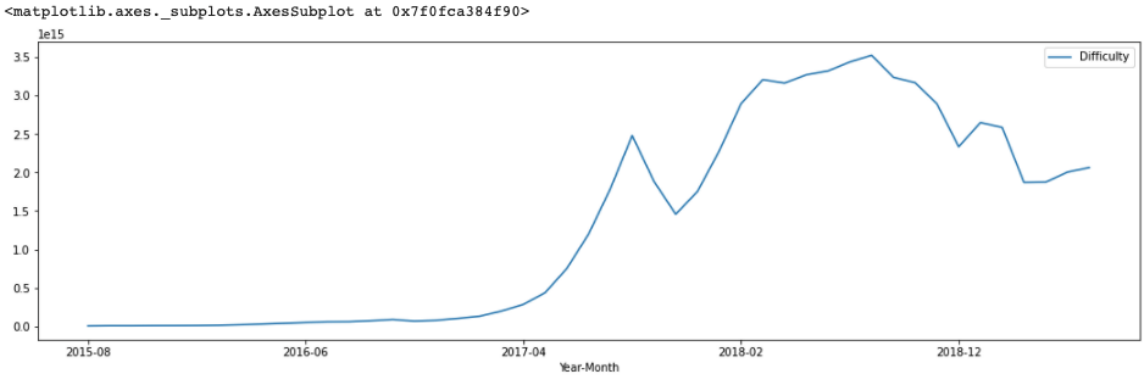
Execution Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_6788

Output: output_gg_part2.csv

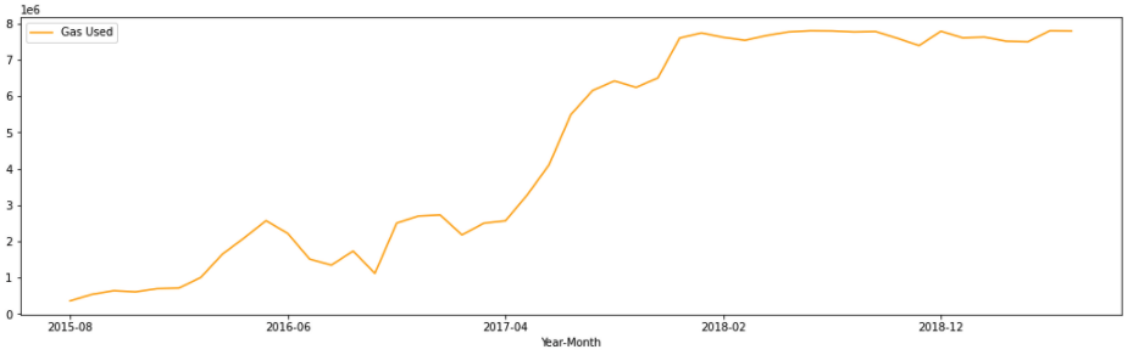
```
('2015-08', (4030960805570.0, 360218.0))
('2015-09', (6577868584193.0, 540131.0))
('2015-10', (6352827787297.0, 641355.0))
('2015-11', (7772752046929.0, 609518.0))
('2015-12', (8279271173937.0, 696716.0))
('2016-01', (9509622515878.0, 714548.0))
('2016-02', (12769672945336.0, 1005720.0))
('2016-03', (20066111443630.0, 1650559.0))
('2016-04', (28654861822851.0, 2097505.0))
('2016-05', (39311237605142.0, 2571039.0))
('2016-06', (49061031594690.0, 2217506.0))
('2016-07', (57786919611082.0, 1513357.0))
('2016-08', (59234534795662.0, 1345600.0))
('2016-09', (72600341869690.0, 1734328.0))
('2016-10', (87658236572825.0, 1117711.0))
('2016-11', (67812116442679.0, 2503838.0))
('2016-12', (77496935205604.0, 2699688.0))
('2017-01', (102282325084568.0, 2727947.0))
('2017-02', (130757112575887.0, 2177629.0))
('2017-03', (198044781241186.0, 2499731.0))
('2017-04', (284126014229797.0, 2568672.0))
('2017-05', (436460724525936.0, 3282000.0))
('2017-06', (752656695470601.0, 4110544.0))
('2017-07', (1200999266684616.0, 5492711.0))
('2017-08', (1790200849201658.0, 6157330.0))
('2017-09', (2481432368327849.0, 6419935.0))
('2017-10', (1885459218688474.0, 6243435.0))
('2017-11', (1457697972100907.0, 6503790.0))
('2017-12', (1753712331569158.0, 7602976.0))
('2018-01', (2281188940495787.0, 7743489.0))
('2018-02', (2895297998304151.0, 7621988.0))
('2018-03', (3208321357874551.0, 7540115.0))
('2018-04', (3163895606602080.0, 7673254.0))
('2018-05', (3271540170064034.0, 7772207.0))
('2018-06', (3322174039186059.0, 7804846.0))
('2018-07', (3438689513774835.0, 7799773.0))
('2018-08', (3525770967637325.0, 7771450.0))
('2018-09', (3239257170470205.0, 7782860.0))
('2018-10', (3168540540592017.0, 7595336.0))
('2018-11', (2893844101806584.0, 7395694.0))
('2018-12', (2335811289311401.0, 7790698.0))
('2019-01', (2650625285026055.0, 7608784.0))
('2019-02', (2586296838944139.0, 7628670.0))
('2019-03', (1871306263893841.0, 7515069.0))
('2019-04', (1875185187990232.0, 7501078.0))
('2019-05', (2006321344431032.0, 7804283.0))
('2019-06', (2064945483489737.0, 7799617.0))
```

Plots:

1. **Difficulty with time**



2. **Gas Used with time**



Explanation:

To compute the requirement of gas with time for contracts; blocks and contracts datasets have been used. First, contracts and blocks dataset are validated and only record with valid data are fetched. From Contracts dataset, block numbers are mapped as key and 1 as the value to count each contract. From Blocks dataset, block number is taken as a key and 'difficulty' and 'gas used' values have been used with a timestamp. Blocks and Contracts data is joined on the key (block number). Resulted data is then mapped by using the timestamp(year-month) as a key and rest of the values as value. Subsequently, reduceByKey() on year-month has been used to sum the values of difficulties, gas_used and counts respectively. Finally, the total difficulty and total gas used are divided by count to get the average of the difficulty and price respectively for a given year-month.

Above graphs implies that, the average gas usage increased steadily until the end of 2017 and reached the peak value. From the start of 2018, there were minor changes observed in the similar range.

On the other hand, complexity (difficulty value) increased exponentially till the Sep 2017 and suddenly observed a decline in the later 2017 and It started to rise again and reached the peak value in Oct 2018. Post which it gradually declined again and couldn't reach the optimum value.

Task 2 - Part 3: *could you correlate the complexity for some of the top-10 contracts found in Part-B by observing the change over their transactions?*

Implemented using Spark

Python Code:

```
import pyspark
import time

'''
Output from Part B

"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"    84155100809965865822726776
"0xfa52274dd61e1643d2205169732f29114bc240b3"    45787484483189352986478805
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"    45620624001350712557268573
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"    43170356092262468919298969
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"    27068921582019542499882877
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"    21104195138093660050000000
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"    15562398956802112254719409
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"    11983608729202893846818681
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"    11706457177940895521770404
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"    83790007519177556240575 00

'''
# Taking random 4 addresses from the above
random_address = ["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", "0xfa52274dd61e1643d2205169732f29114bc240b3", "0x6fc82a5fe25a5cdb
global curr_address

def check_good_transaction(line):
    try:
        fields = line.split(',')
        if len(fields) != 7:
            return False

        if fields[2] == curr_address:
            return True
    except:
        return False

def check_good_blocks(line):
    try:
        fields = line.split(',')
        if len(fields) != 9:
            return False
        float(fields[0]) # block number
        float(fields[6]) # gas used
        float(fields[7]) # timestamp
        return True
    except:
        return False


for current_address in random_address:
    # Create Spark context object
    curr_address = current_address
    sc = pyspark.SparkContext()
    print(sc.applicationId)

    # Reading Transactions Dataset
    transactions_data = sc.textFile('/data/ethereum/transactions')
    valid_transactions_data = transactions_data.filter(check_good_transaction)
    transaction_block_number = valid_transactions_data.map(lambda l: (l.split(',')[0], l.split(',')[2]))
    # print(transaction_block_number.take(2))
    # [(u'2280145', u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444'), (u'2280145', u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444')]

    blocks_data = sc.textFile('/data/ethereum/blocks')
    valid_blocks_data = blocks_data.filter(check_good_blocks)

    blocks_mapper = valid_blocks_data.map(lambda l: (l.split(',')[0], (int(l.split(',')[3]), time.strftime("%Y-%m", time.gmtime(f
    print(blocks_mapper.take(2))
    # [(u'4776199', (1765656009004680, '2017-12')), (u'4776200', (1765656009037448, '2017-12'))]

    results1 = blocks_mapper.join(transaction_block_number)
    # print(results.take(2))
    # [(u'5003843', ((2604480603800323, '2018-01'), u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444')), (u'2114577', ((6406150115035

    step1 = results1.map(lambda x: (x[1][0][1], x[1][0][0]))
    # print(step1.take(2))
    # [('2017-09', 2925691023921253), ('2017-12', 1574664440410841)]

    step2 = step1.reduceByKey(lambda a,b: float(a+b)).sortByKey(ascending=True)
    # print(step2.take(2))
    # [('2016-07', 2.063280508261113e+16), ('2016-08', 1.9618904264539485e+17)]

    step2.saveAsTextFile('Result_address_'+ current_address)

    sc.stop()

```

Explanation:

In part B, top 10 smart contracts were extracted, out of which random 4 addresses have been used for the analysis. These 4 addresses have been added in a list. For loop is used to run the code for all the addresses. For each address id, first validating the blocks and transactions data and extracting the transactions for the current address only. Block number for these transactions is used as a key to join the blocks data with the block_number key to get the difficulty value and timestamp value.

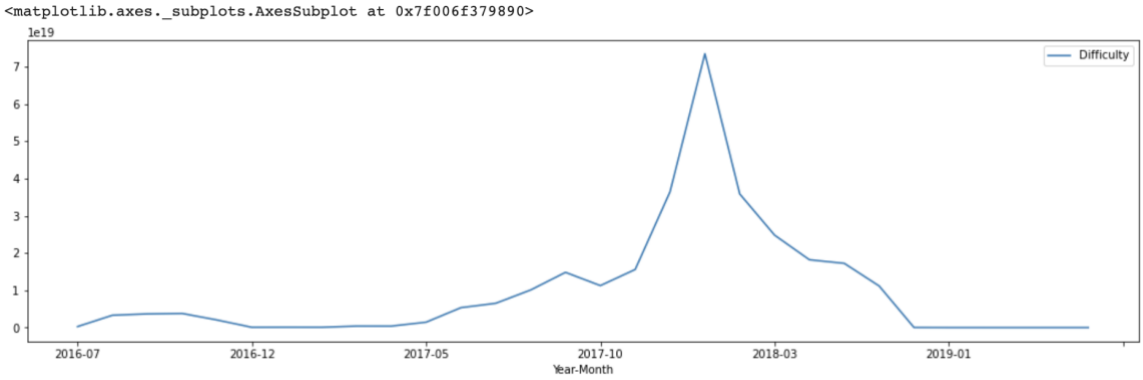
Resulted dataset is mapped on key timestamp (Year-Month) and difficulty as the value. It is reduced by the key to sum the difficulty values for each year-month. These results are extracted as csv and plotted using ipynb notebook, attached with the code.

Execution Job ID:

1. **For Address: "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444" → application_1649894236110_1714**

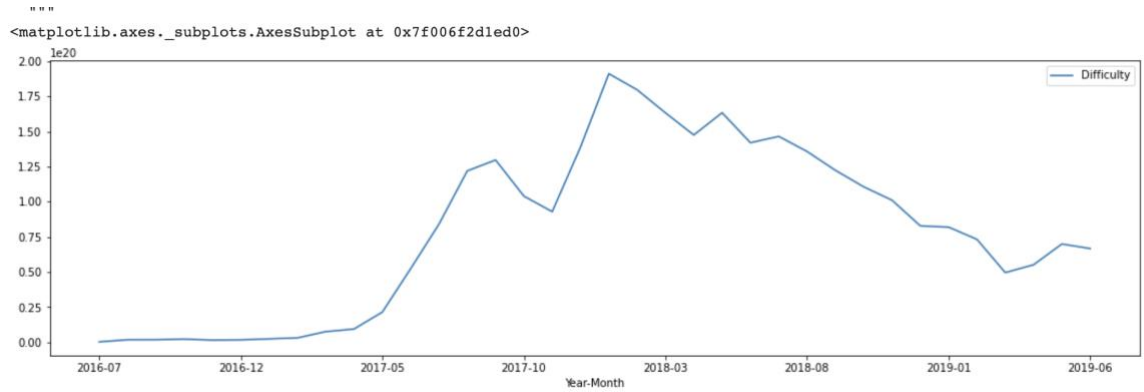
   Result file: *Result_address_0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444.csv*

   Plot: Complexity vs Time



2. **For address: "0xfa52274dd61e1643d2205169732f29114bc240b3"→ application_1649894236110_1732**

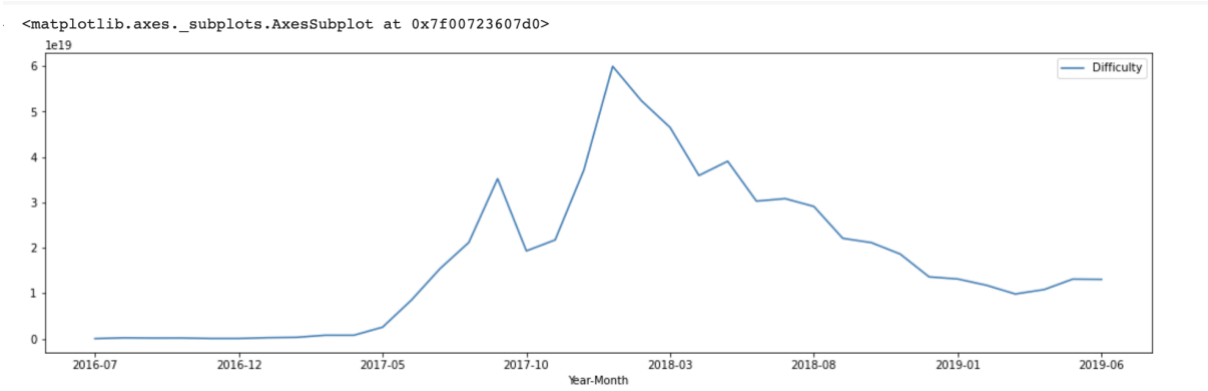
   Result file: *Result_address_0xfa52274dd61e1643d2205169732f29114bc240b3.csv*
   Plot: Complexity vs Time



3. **For address: "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"→**

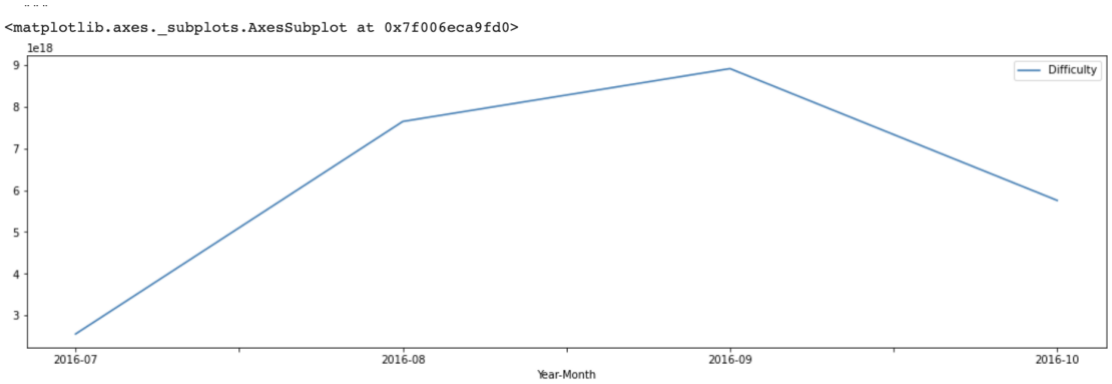
application_1649894236110_1743

Result file: *Result_address_0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8.csv*]
Plot: Complexity vs Time



4. **for address: "0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"→ application_1649894236110_1758**

Result file: *Result_address_0xbfc39b6f805a9e40e77291aff27aee3c96915bdd.csv*
Plot: Complexity vs Time



Plot Results:

From the above graphs, we can observe that in first three cases complexity first increased and reaches the peak value in last quarter of 2017 and first quarter of 2018 and then gradually decreases which can correlated to the overall result as observed in the above part. In the fourth case, we can see there are only 4 records where the trend is same where value first increases and reaches a peak and decreases again. Therefore, it can be concluded that the difficulty is correlated with time, and it increases first reaches the peak value in later 2017 and early 2018 and then decreases.

Task 3: Comparative Analysis

Spark Code: compare_partb_spark.py

```python

import pyspark
from time import *

def check_good_transactions(line):
    """ Function to check if the transactions record is valid"""
    try:
        fields = line.split(",")
        if len(fields) == 7:
            int(fields[3])
            return True
        else:
            return False
    except:
        return False

def check_good_contracts(line):
    """ Function to check if the contracts record is valid"""
    try:
        fields = line.split(",")
        if len(fields) == 5:
            fields[0]
            return True
        else:
            return False
    except:
        return False

# Storing Start Time
start_time = time()

# Spark Context object
sc = pyspark.SparkContext()
print(sc.applicationId)

# Read Transactions Dataset
transaction_data = sc.textFile("/data/ethereum/transactions")

# Validate transactions dataset and take only valid data
valid_transaction_data = transaction_data.filter(check_good_transactions).map(lambda x: x.split(","))

# Map to_address and value
transaction_features = valid_transaction_data.map(lambda x: (x[2], int(x[3])))
result1 = transaction_features.reduceByKey(lambda a,b: a+b)

# Read Contracts Dataset
contracts_data = sc.textFile("/data/ethereum/contracts")
# Validate Contracts datasset and take only valid data
valid_contracts_data = contracts_data.filter(check_good_contracts).map(lambda x: x.split(","))

# Fetch addresses from Contracts as these will be considered as smart contracts
contracts_features = valid_contracts_data.map(lambda x: (x[0], None))

result2 = result1.join(contracts_features)
# (address, (summed_value, None))

top10_data = result2.takeOrdered(10, key = lambda x: -x[1][0])
top10_data = sc.parallelize(top10_data).saveAsTextFile("output_compare_sparkresult2")

# Storing stop time
stop_time = time()
print("Total Time taken by Spark Job: {}".format(stop_time - start_time))

```

Spark Output:

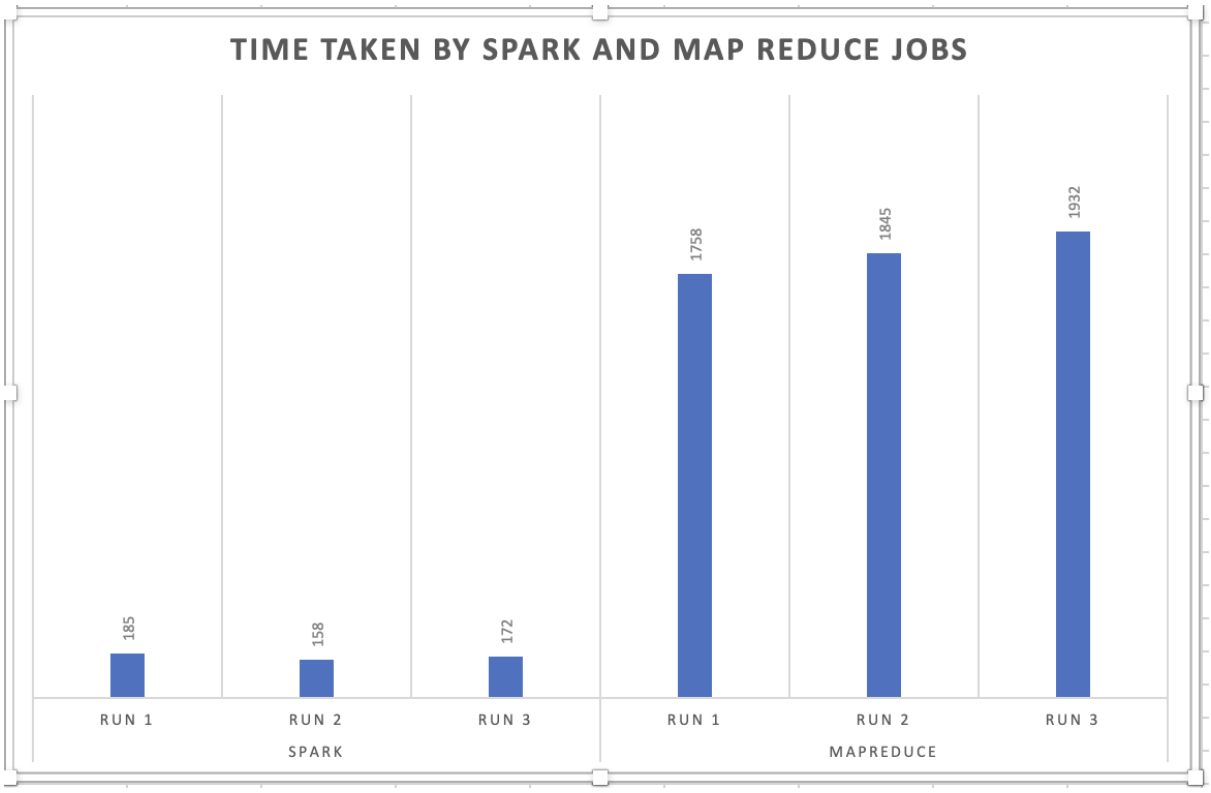
```
(u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', (84155100809965865822726776L, None))
(u'0xfa52274dd61e1643d2205169732f29114bc240b3', (45787484483189352986478805L, None))
(u'0x7727e5113d1d161373623e5f49fd568b4f543a9e', (45620624001350712557268573L, None))
(u'0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', (43170356092262468919298969L, None))
(u'0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8', (27068921582019542499882877L, None))
(u'0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', (21104195138093660050000000L, None))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', (15562398956802112254719409L, None))
(u'0xbb9bc244d798123fde783fcc1c72d3bb8c189413', (11983608729202893846818681L, None))
(u'0xabbb6bebfa05aa13e908eaa492bd7a8343760477', (11706457177940895521770404L, None))
(u'0x341e790174e3a4d35b65fdc067b6b5634a61caea', (83790007519177556240575O0L, None))
```

Performance Comparison:

Spark		
Run	**Execution Job ID**	**Time Taken**
1	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7308	185s
2	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7324	158s
3	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7372	172s

Map Reduce		
Run	**Execution Job ID**	**Time Taken**
1	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7412/	1758s
1	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7526/	
2	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7769/	1845s
2	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7868/	
3	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7903/	1932s
3	http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1648683650522_7996/	

Plots:



Explanation:
Above performance of Spark and Map Reduce jobs for the same operation implies that the Spark jobs are much faster than the MapReduce jobs.

Reason for high performance in Spark is because it used RDDs to store intermediate results and uses the same for further transformations. Whereas, in case of Hadoop, Map Reduce jobs write the intermediate results to HDFS file system which is an expensive operation. Further, shuffle and sort in

Map Reduce jobs take additional time for large datasets. Hence, the performance of the Map Reduce jobs is poor as compared to Spark.

For the given dataset, Spark is well suited as compared to MapReduce.

Task 4: Fork The Chain

There have been several forks of Ethereum in the past. Identify one or more of these and see what effect it had on price and general usage. For example, did a price surge/plummet occur, and who profited most from this?

Implemented using Spark

Python Code: ForkTheChain.py

PART_D > ForkTheChain > ForkTheChain.py

```python
import pyspark
import datetime
import time

# Sample Forks Identified in the given timeframe
fork_date_and_name = [[datetime.datetime(2016,3,15),'Homestead'], [datetime.datetime(2016,11,23),'Spurious Dragon

def check_good_line(line):
    """
    Function to get valid records and check if the record timstamp is within the range of 7 days prior and after
    """
    try:
        fields = line.split(',')

        block_time = datetime.datetime.fromtimestamp(int(fields[6]))
        block_date = time.gmtime(float(fields[6]))
        address = fields[2]
        gas_value = int(fields[5])

        fork_record = False

        for fork_data in fork_date_and_name:
            start_date = fork_data[0] + datetime.timedelta(-7) # Start Date 7 days prior the fork date
            end_date = fork_data[0] + datetime.timedelta(7) # End Days 7 days after the fork date

            if (block_time > start_date) and (block_time < end_date):
                fork_name = fork_data[1] # Fetch Current fork name
                fork_record = True
                break
        if fork_record:
            return True

    except:
        return False


def mapper(line):
    # mapper function to apply on data to get the required timestamp data only
    try:
        fields = line.split(',')

        block_time = datetime.datetime.fromtimestamp(int(fields[6]))
        block_date = time.gmtime(float(fields[6]))
        address = fields[2]
        gas_value = int(fields[5])

        fork_record = False

        for fork_data in fork_date_and_name:
            start_date = fork_data[0] + datetime.timedelta(-7) # Start Date 7 days prior the fork date
            end_date = fork_data[0] + datetime.timedelta(7) # End Days 7 days after the fork date

            if (block_time > start_date) and (block_time < end_date):
                fork_name = fork_data[1] # Fetch Current fork name
                fork_record = True
                break
        if fork_record:
            full_date = str(block_date.tm_year)+'-'+str(block_date.tm_mon)+'-'+str(block_date.tm_mday)
            return ((full_date, address), (gas_value, fork_name, 1))
    except:
        pass
```

```
62
63
64    # Create Spark Context Object
65    sc = pyspark.SparkContext()
66    print(sc.applicationId)
67
68    # Read Transactions dataset
69    transactions_data = sc.textFile('/data/ethereum/transactions')
70    valid_transactions_data = transactions_data.filter(check_good_line)
71
72    # Apply mapper function to get data in the range of forks
73    step1 = valid_transactions_data.map(mapper)
74    print(step1.take(2))
75
76    # Map ton key => fork name and date and values as gas value and counter
77    step2 = step1.map(lambda x: ((x[1][1], x[0][0]), (x[1][0], x[1][2])))
78    print(step2.take(2))
79
80    # Add Gas values and counter by reducing on key
81    step3 = step2.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1]))
82
83    # Map on fork name as key and average gas value as value for each date
84    step4 = step3.map(lambda x: (x[0], float(x[1][0] / x[1][1]))).sortByKey(ascending = True)
85
86    # Save output
87    step4.saveAsTextFile('output_forkthechain')
88
```

Execution Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_2374

Execution Command: *spark-submit ForkTheChain.py*

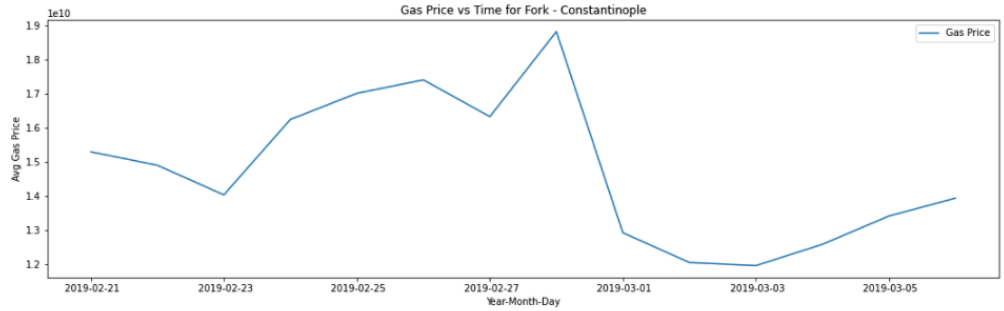
Output: output_forkthechain.csv

```
(('Byzantium', '2017-10-10'), 25058292801.0)
(('Byzantium', '2017-10-11'), 30039329444.0)
(('Byzantium', '2017-10-12'), 24512657108.0)
(('Byzantium', '2017-10-13'), 26489325565.0)
(('Byzantium', '2017-10-14'), 24196653845.0)
(('Byzantium', '2017-10-15'), 23514174126.0)
(('Byzantium', '2017-10-16'), 16678649825.0)
(('Byzantium', '2017-10-17'), 14194850634.0)
(('Byzantium', '2017-10-18'), 14081137033.0)
(('Byzantium', '2017-10-19'), 12100849868.0)
(('Byzantium', '2017-10-20'), 14610296315.0)
(('Byzantium', '2017-10-21'), 12128683877.0)
(('Byzantium', '2017-10-22'), 10666208189.0)
(('Byzantium', '2017-10-8'), 21051671916.0)
(('Byzantium', '2017-10-9'), 24174030452.0)
(('Constantinople', '2019-2-21'), 15301279312.0)
(('Constantinople', '2019-2-22'), 14906500367.0)
(('Constantinople', '2019-2-23'), 14038665092.0)
(('Constantinople', '2019-2-24'), 16250710476.0)
(('Constantinople', '2019-2-25'), 17016748385.0)
(('Constantinople', '2019-2-26'), 17411522198.0)
(('Constantinople', '2019-2-27'), 16333133968.0)
(('Constantinople', '2019-2-28'), 18828795385.0)
(('Constantinople', '2019-3-1'), 12927381010.0)
(('Constantinople', '2019-3-2'), 12055153413.0)
(('Constantinople', '2019-3-3'), 11963576046.0)
(('Constantinople', '2019-3-4'), 12586925298.0)
(('Constantinople', '2019-3-5'), 13416451614.0)
(('Constantinople', '2019-3-6'), 13941269746.0)
(('Homestead', '2016-3-10'), 30204433046.0)
(('Homestead', '2016-3-11'), 29035652694.0)
(('Homestead', '2016-3-12'), 29197411911.0)
(('Homestead', '2016-3-13'), 30820960382.0)
(('Homestead', '2016-3-14'), 28509995012.0)
(('Homestead', '2016-3-15'), 26342144976.0)
(('Homestead', '2016-3-16'), 25523949148.0)
(('Homestead', '2016-3-17'), 27708932990.0)
(('Homestead', '2016-3-18'), 26302902479.0)
(('Homestead', '2016-3-19'), 29326753003.0)
(('Homestead', '2016-3-20'), 26655411712.0)
(('Homestead', '2016-3-21'), 26198708914.0)
(('Homestead', '2016-3-8'), 53595079870.0)
(('Homestead', '2016-3-9'), 38089619701.0)
(('Spurious Dragon', '2016-11-16'), 24744831659.0)
(('Spurious Dragon', '2016-11-17'), 24333883462.0)
(('Spurious Dragon', '2016-11-18'), 24779293462.0)
(('Spurious Dragon', '2016-11-19'), 24367176414.0)
(('Spurious Dragon', '2016-11-20'), 24297865024.0)
(('Spurious Dragon', '2016-11-21'), 24827550548.0)
(('Spurious Dragon', '2016-11-22'), 23328942477.0)
(('Spurious Dragon', '2016-11-23'), 24571404784.0)
(('Spurious Dragon', '2016-11-24'), 27957005767.0)
(('Spurious Dragon', '2016-11-25'), 17357950828.0)
(('Spurious Dragon', '2016-11-26'), 24476330142.0)
(('Spurious Dragon', '2016-11-27'), 25187618136.0)
(('Spurious Dragon', '2016-11-28'), 25725607641.0)
(('Spurious Dragon', '2016-11-29'), 25162584778.0)
```

Plots:

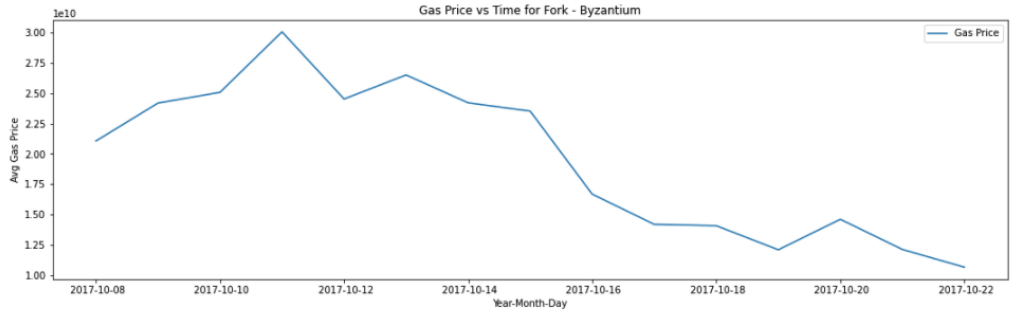
1. Constantinople Fork – Date: 28/2/2019

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2dad48e1d0>
```



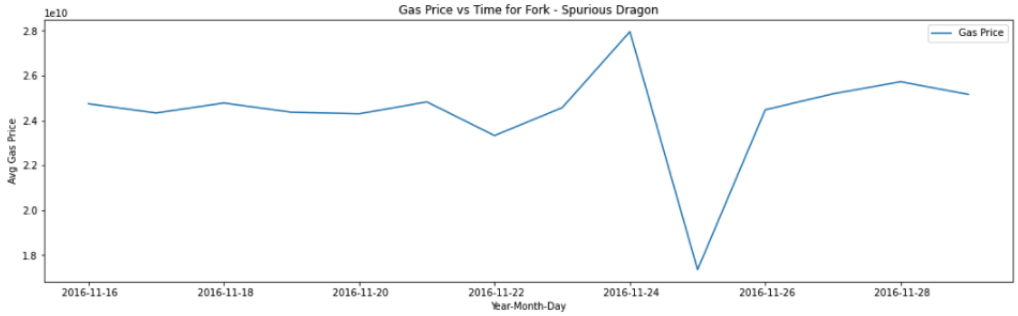
2. Byzantium Fork – Date: 16/10/2017

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2dacf47f90>
```



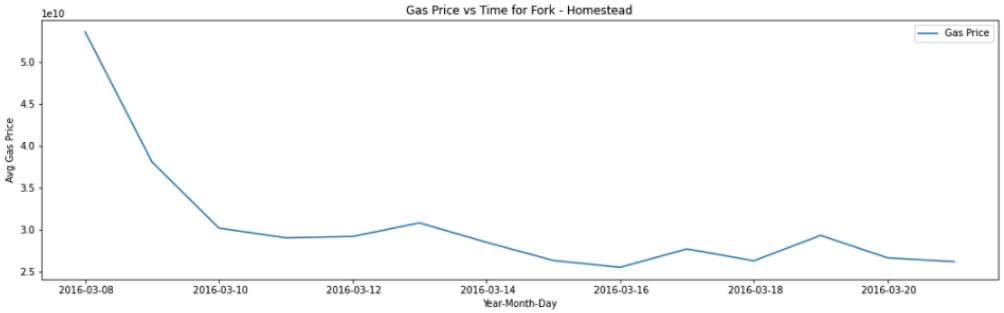
3. Spurious Dragon – Date: 23/11/2016

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2dacdda910>
```



4. Homestead – Date: 15/3/2016

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2dacdbc790>
```



Explanation:
Ethereum data has been provided from 2016 to June 2019. Forks have been identified online in the given time-period online. There are number of forks occurred, out of which sample 4 forks have been used for the given problem. First, we have taken all the four forks date and name in a list of lists. 'Transactions' dataset is then read and validated against these dates. Transactions occurred 7 days prior or 7 days after the fork dates have been extracted for all 4 forks by using the datetime comparison. Validated data is then mapped based on full date and address as key and the fork names and count as values. Resulted dataset is then mapped by using the combination of Fork Name and Date as the key to identify gas price and count as values.y
In the next step, reduceByKey is used to sum all the gas price for each Fork name on a given day. Finally, Average gas price for each day is calculated.

Ideology behind this approach is that the fork should have an impact on the gas price from the day of fork. Therefore, before and after data is identified and plotted.

Plots Result:
In the first example, **Constantinople** which occurred on 28/2/2019, it can be observed that the avg gas price increased and reached the peak value in 14 days and there is a steep downfall in the average gas price after 28/2/2019.

In the case of **Byzantium** which occurred on 16/10/2017, it can be observed that there is a continuous decline in the average gas price after the fork for next 3 days.

In case of **Spurious Dragon** which occurred on 23/11/2016, the gas price decreased drastically and started recovering again after 2 days of the fork.

Finally, In case of **Homestead** which occurred on 15/3/2016, the gas price slightly decreased but fluctuated for next few days.

Thus, we can conclude that, **the forks had a plummet impact on the gas price** of the transactions. This means that the **top 10 transactions will gain the most profit** as the gas price goes down for the transactions, so each transaction will provide the profit in terms of cost.

Task 5 – Wash Trading

Wash trading is defined as "Entering into or purporting to enter into, transactions to give the appearance that purchases, and sales have been made, without incurring market risk or changing the trader's market position" Unregulated exchanges use these to fake up to 70% of their trading volume? Which addresses are involved in wash trading? Which trader has the highest volume of wash trades? How certain are you of your result? More information can be found at https://dl.acm.org/doi/pdf/10.1145/3442381.3449824. One way to measure ether balance over time is also possible but you will need to discuss accuracy concerns.

Implemented using Spark

Python Code: washtrading.py

```python
import pyspark
import time
import pyspark.sql.functions as F


def check_good_lines(line):
    """
    Function to validate Transactions Dataset
    """
    try:
        fields = line.split(',')
        if len(fields) != 7:
            return False

        float(fields[3])
        return True
    except:
        return False

# Create Spark Context Object
sc = pyspark.SparkContext()
print(sc.applicationId)

#reading the etherum data from the HDFS
transactions_data = sc.textFile("/data/ethereum/transactions")

#fvalidating the transactions dataset
valid_transactions_data = transactions_data.filter(check_good_lines)

# Scenario: Self Trade

# RDD with required values
requiredtransactions = valid_transactions_data.map(lambda x: (x.split(',')[1], x.split(',')[2],  x.split(',')[3],  x.split(',')[6]))

columns = ['from_address', 'to_address', 'value',' timestamp']
# Create Dtaaframe from RDD and give column names using columns list
df = requiredtransactions.toDF(columns)

# Using sql functions to get rows with same column values.
df1 = df.filter(F.col('from_address') == F.col('to_address'))

# Create RDD with Key as from and to address and value as value
self_transaction_rdd = df1.rdd.map(lambda x: ((x[0],x[1]), float(x[2])))
# print(self_transaction_rdd.take(3))
# [((u'0x7ed1e469fcb3ee19c0366d829e291451be638e59', u'0x7ed1e469fcb3ee19c0366d829e291451be638e59'), (u'9650000000000000', u'1474189901')), ((

# RDD to sum values for same from and to address
result = self_transaction_rdd.reduceByKey(lambda a, b: a+b)


 # fetch top 10 trades by value
 top10_data = result.takeOrdered(10, key = lambda x: -x[1])
 top10_data = sc.parallelize(top10_data).saveAsTextFile("wash_trades_top10")

```

Execution Job ID:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1649894236110_2736

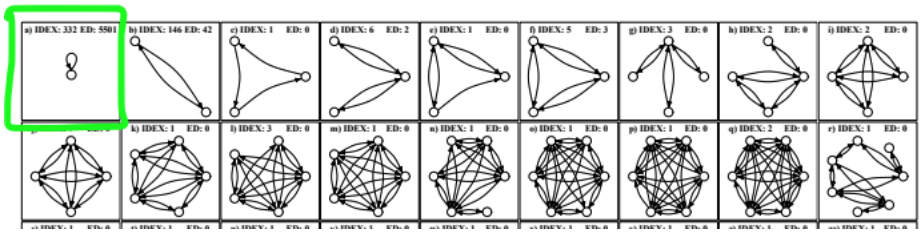
Execution command: *spark-submit washtrading.py*

Output: wash_trades_top10.txt

PART_D > WashTrading > wash_trades_top10.txt

```
((u'0x02459d2ea9a008342d8685dae79d213f14a87d43', u'0x02459d2ea9a008342d8685dae79d213f14a87d43'), 1.9548531332493176e+25)
((u'0x32362fbfff69b9d31f3aae04faa56f0edee94b1d', u'0x32362fbfff69b9d31f3aae04faa56f0edee94b1d'), 5.295490520134211e+24)
((u'0x0c5437b0b6906321cca17af681d59baf60afe7d6', u'0x0c5437b0b6906321cca17af681d59baf60afe7d6'), 2.3771525723546667e+24)
((u'0xdb6fd484cfa46eeeb73c71edee823e4812f9e2e1', u'0xdb6fd484cfa46eeeb73c71edee823e4812f9e2e1'), 4.1549736829070815e+23)
((u'0xd24400ae8bfebb18ca49be86258a3c749cf46853', u'0xd24400ae8bfebb18ca49be86258a3c749cf46853'), 2.2700012958e+23)
((u'0xa9c7d31bb1879bff8be25ead2f59b310a52b7c5a', u'0xa9c7d31bb1879bff8be25ead2f59b310a52b7c5a'), 1.6966220991798057e+23)
((u'0x6cc5f688a315f3dc28a7781717a9a798a59fda7b', u'0x6cc5f688a315f3dc28a7781717a9a798a59fda7b'), 1.575893013642e+23)
((u'0x5b76fbe76325b970dbfac763d5224ef999af9e86', u'0x5b76fbe76325b970dbfac763d5224ef999af9e86'), 7.873327492788825e+22)
((u'0xdd3e4522bdd3ec68bc5ff272bf2c64b9957d9563', u'0xdd3e4522bdd3ec68bc5ff272bf2c64b9957d9563'), 5.790175685075673e+22)
((u'0x005864ea59b094db9ed88c05ffba3d3a3410592b', u'0x005864ea59b094db9ed88c05ffba3d3a3410592b'), 3.7199e+22)
```

Explanation: There are multiple scenarios for Wash Trading. One of them is the self-transaction which is the most common type of trade. Here, from_address is equal to the to_address in the transaction. Reference for this trade has been added below.



For this, transaction dataset has been used and validated. RDD is created by using the required columns only. RDD is converted into dataframe with appropriate column names. Pyspark SQL functions are used to filter the dataframe where column from_address value is same as column to_address. This gives us all the self-trades with their respective values. Resulting dataframe is again converted into a RDD and mapped on key as from and to address and 'value' as the value. These values are summed using reduceByKey method on from and to address as key to get the total value of trades in self-trade wash trading.

Finally, top 10 self-trades have been fetched using takeOrdered function.

Highest volume self-trade wash-trader is with address - 0x02459d2ea9a008342d8685dae79d213f14a87d43

Reference for the research paper: https://arxiv.org/pdf/2102.07001.pdf

6.1 Wash Trading Structures

All trades of the candidate set SCCs were labeled with our volume matching algorithm. As a result, we can study the structures that consist of identified wash trades. Figure 6 illustrates the result for both IDEX and EtherDelta, where the counts indicate how many variants with different traders have been observed on each DEX. On EtherDelta, the vast majority of wash trades are performed as self-trades (a)), where an account is able to trade with itself, which could easily be prevented. Although also common, they do not appear to the same extent on IDEX. Second only to the self-trades, the structure consisting of two accounts (b)) is the next simplest, and at least on IDEX it is also quite common. More complex structures consisting of three or more accounts are mainly found on IDEX. In almost every advanced structure, several short sub-cycles can be found. In some instances the wash trade subgraphs are completely connected. The complex structures consisting of simpler sub-structures may indicate that wash trading is performed in simple cycles on a low-level, but that actors make an effort to hide these among multiple trading accounts. Branched structures with sub-cycles of length greater than three do not occur here.