BIG DATA PROCESSING

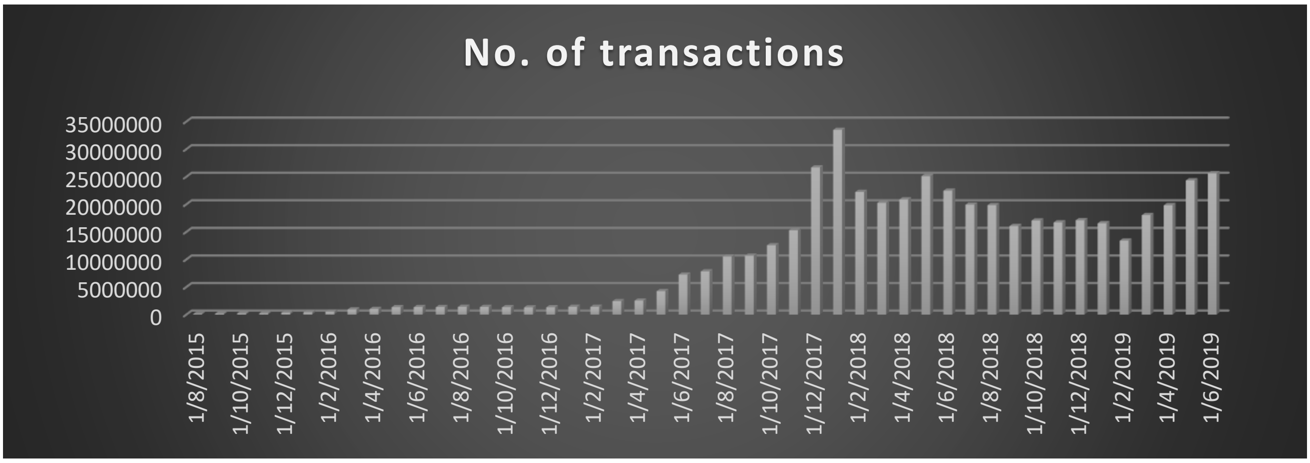
COURSEWORK: ETHEREUM ANALYSIS

ANALYSIS OF ETHEREUM TRANSACTIONS AND SMART CONTRACTS

Merkouriadis Konstantinos

PART A: TIME ANALYSIS (Number of Transactions)

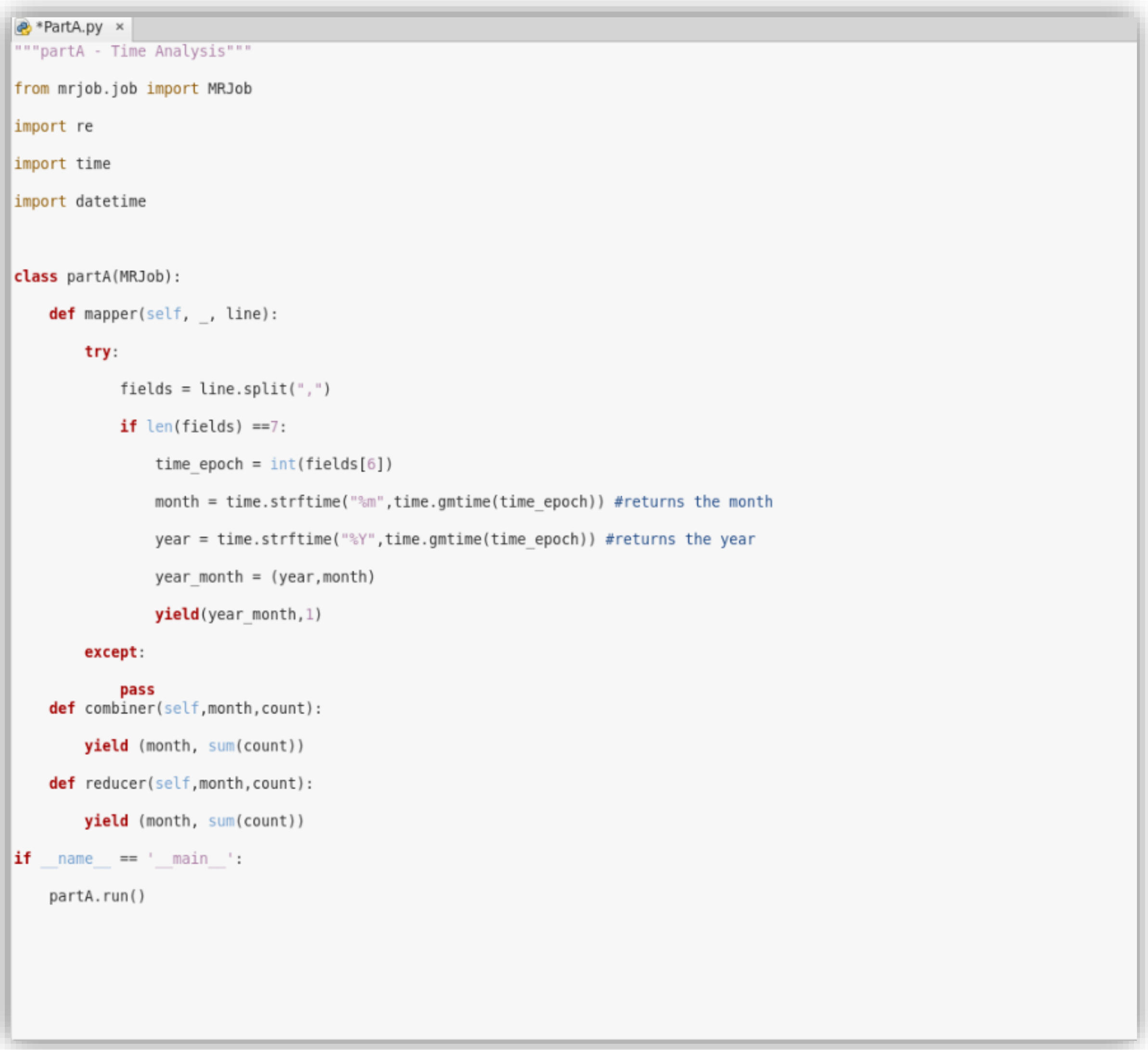
Approach: The block timestamp field in the transaction dataset is used to measure the number of transactions happening per month. The combined value(sum) of all monthly timestamp values returns the total number of transactions that occurred.



Graph: The above Bar Plot was plotted using Excel. The bar chart displays the number of Ethereum transactions recorded from August 2015 to June 2019 for each month. The bar chart shows little to no positive association from the start of Ethereum until February 2017, save for a small rise in May 2016. Then there is a huge rise from May 2017 to January 2018, which is also the bar chat's highest point. The outcome then differs, displaying a poor average correlation from the highest peak until February 2019, and then a strong positive correlation until June 2019. The high Ethereum variance is seen by this constant shift in correlation.

Job id: http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_2736/

Code: The Hadoop code to create the plot is shown below. Program name is PartA.py



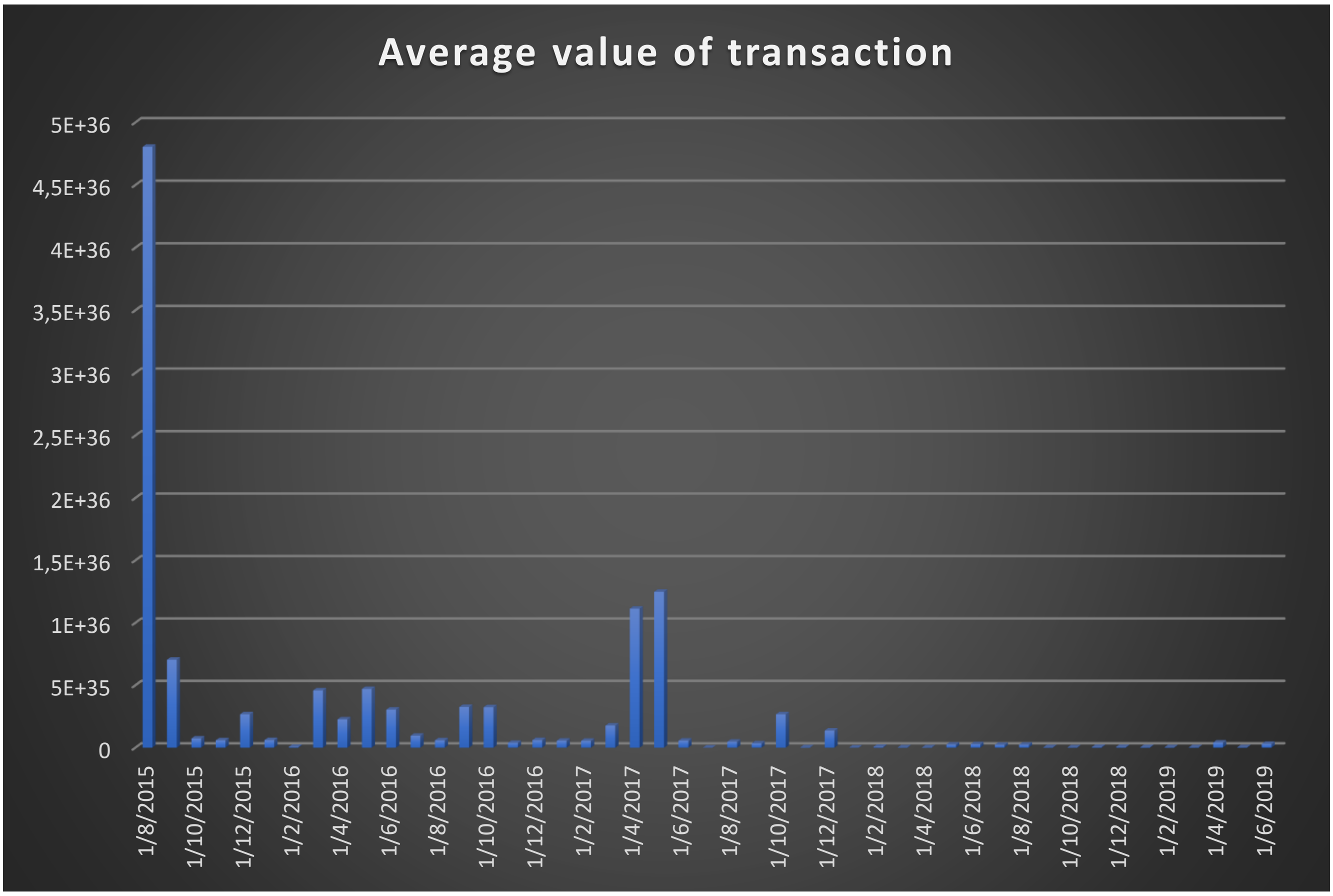
```
"""partA - Time Analysis"""

from mrjob.job import MRJob

import re

import time

import datetime


class partA(MRJob):

    def mapper(self, _, line):

        try:

            fields = line.split(",")

            if len(fields) ==7:

                time_epoch = int(fields[6])

                month = time.strftime("%m",time.gmtime(time_epoch)) #returns the month

                year = time.strftime("%Y",time.gmtime(time_epoch)) #returns the year

                year_month = (year,month)

                yield(year_month,1)

        except:

            pass
    def combiner(self,month,count):

        yield (month, sum(count))

    def reducer(self,month,count):

        yield (month, sum(count))

if __name__ == '__main__':

    partA.run()
```

This implementation makes use of the program MapReduce. Three functions exist: mapper, combiner, and reducer. To avoid malformed lines, the mapper feature checks whether the line has been properly formatted. The attempt and error construct are then used to capture any possible exception. If the lines are formatted correctly, then the data, in this case time, which is the 6th element of the array, fields, is stored on the time-epoch variable and emitted with a value of 1.

As they run on each partition, the combiner and reducer function are quite similar. The role of the combiner, however, serves as a preliminary reducer.

Even though the combiner feature is optional, minimizing the number of items generated is very beneficial for improving performance.

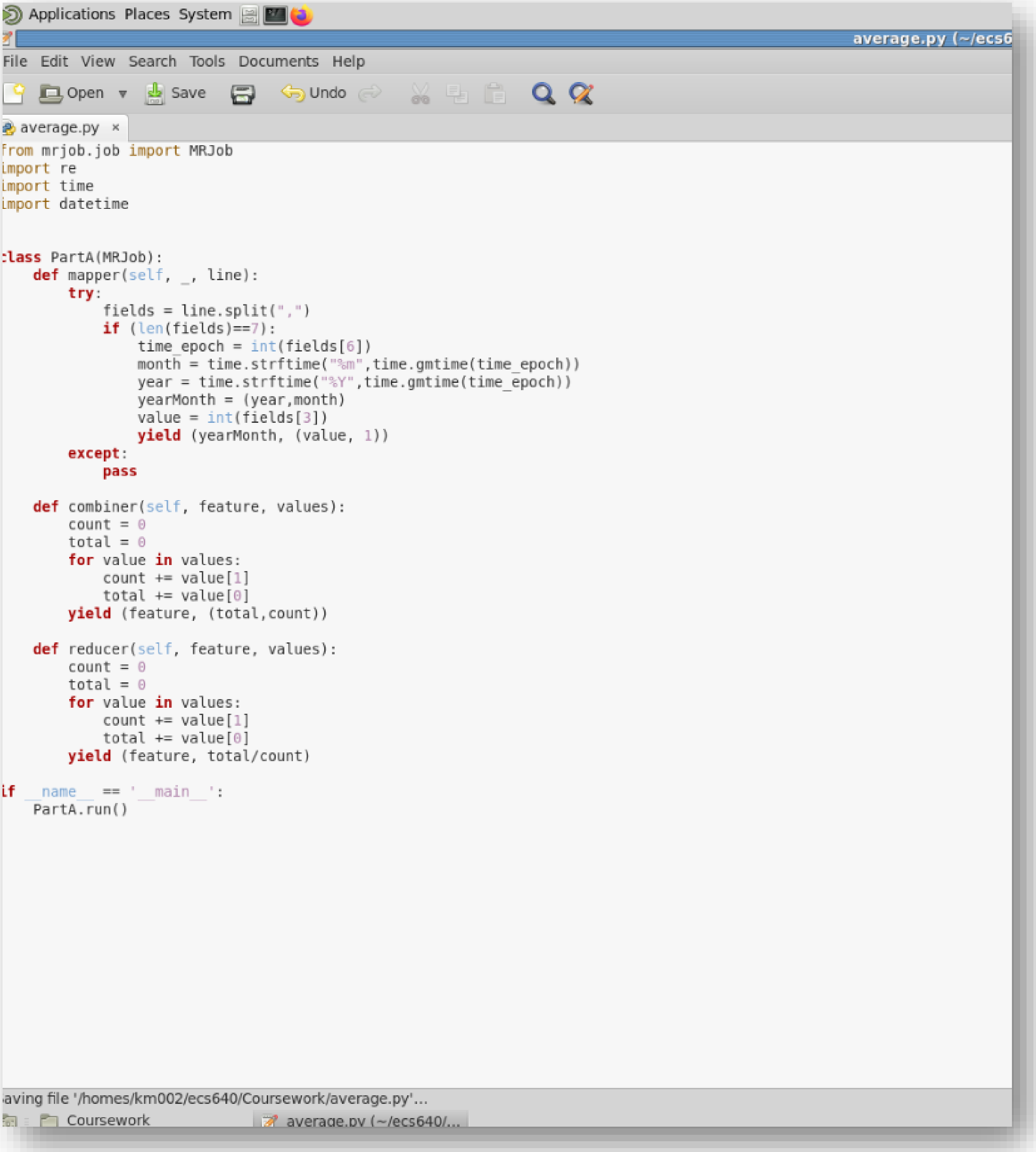
PART A: TIME ANALYSIS (Average value of Transactions)



Graph: The above Bar Plot was plotted using Excel. The bar chart displays the average value of Ethereum transactions recorded from August 2015 to June 2019 for each month.

Job id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_2998/

Code: The Hadoop code to create the plot is shown below. Program name is average.py


Applications Places System
average.py (~/ecs6
File Edit View Search Tools Documents Help
Open
Save
Undo
average.py ×

```
from mrjob.job import MRJob
import re
import time
import datetime


class PartA(MRJob):
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if (len(fields)==7):
                time_epoch = int(fields[6])
                month = time.strftime("%m",time.gmtime(time_epoch))
                year = time.strftime("%Y",time.gmtime(time_epoch))
                yearMonth = (year,month)
                value = int(fields[3])
                yield (yearMonth, (value, 1))
        except:
            pass

    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total,count))

    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, total/count)

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

aving file '/homes/km002/ecs640/Coursework/average.py'...

This implementation makes use of the MapReduce software. There are three functions: mapper, combiner, and reducer. The mapper function tests if the line has been correctly formatted to prevent malformed lines. Any conceivable exception is then caught using the attempt and error construct. If the lines are correctly formatted, then the data fields, in this case time, which is the sixth element of the array, are stored on the time-epoch vector and emitted with a value of 1.

The combiner and reducer functions are very similar as they operate on each partition. However, the function of the combiner serves as a preliminary reducer. Although the combiner function is optional, it is very useful for optimizing efficiency to minimize the number of items produced.

PART B: TOP TEN MOST POPULAR SERVICES

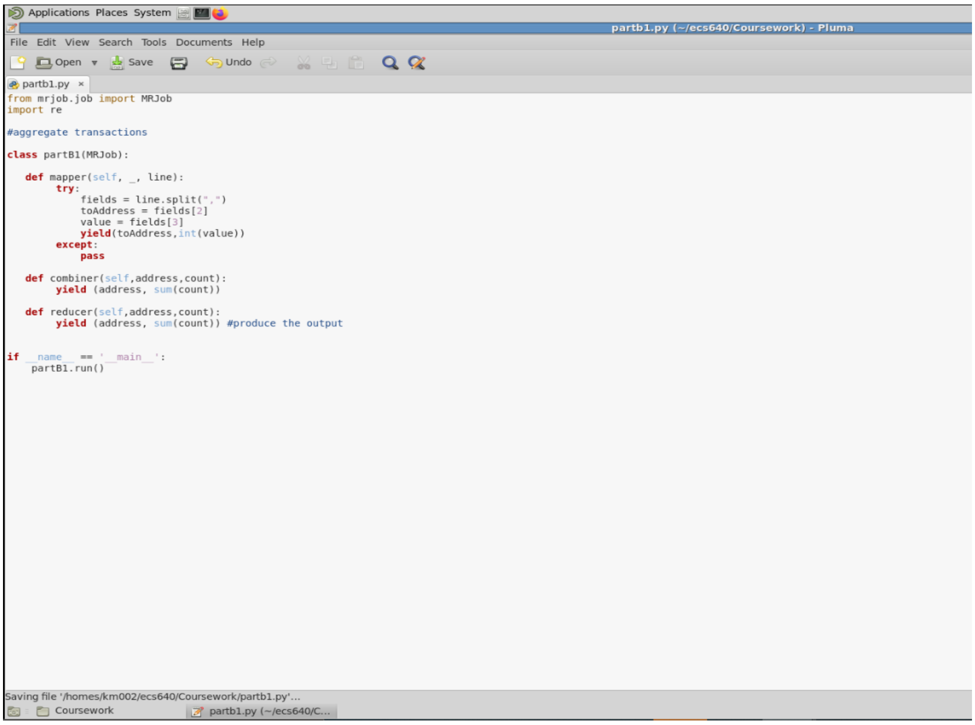
JOB 1 - INITIAL AGGREGATION

In order to figure out which services are the most common, you must first aggregate transactions to see how often each address has been involved in the user space. In the to-address field, you want to sum value for addresses. This is going to be close to the word count we saw in Lab 1 and Lab 2.

Approach: As given, the value field is aggregated for addresses in the to address field from the transaction file to determine the aggregation of transactions.

Code: Hadoop Map/Reduce program to calculate the initial aggregation is given below. Program name is Partb1.py.

Job id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_3021/


Applications Places System
partb1.py (~/ecs640/Coursework) - Pluma
File Edit View Search Tools Documents Help
Open
Save
Undo
partb1.py

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

#aggregate transactions

class partB1(MRJob):

   def mapper(self, _, line):
        try:
            fields = line.split(",")
            toAddress = fields[2]
            value = fields[3]
            yield(toAddress,int(value))
        except:
            pass

   def combiner(self,address,count):
        yield (address, sum(count))

   def reducer(self,address,count):
        yield (address, sum(count)) #produce the output


if __name__ == '__main__':
    partB1.run()
```

```
Saving file '/homes/km002/ecs640/Coursework/partb1.py'...
```

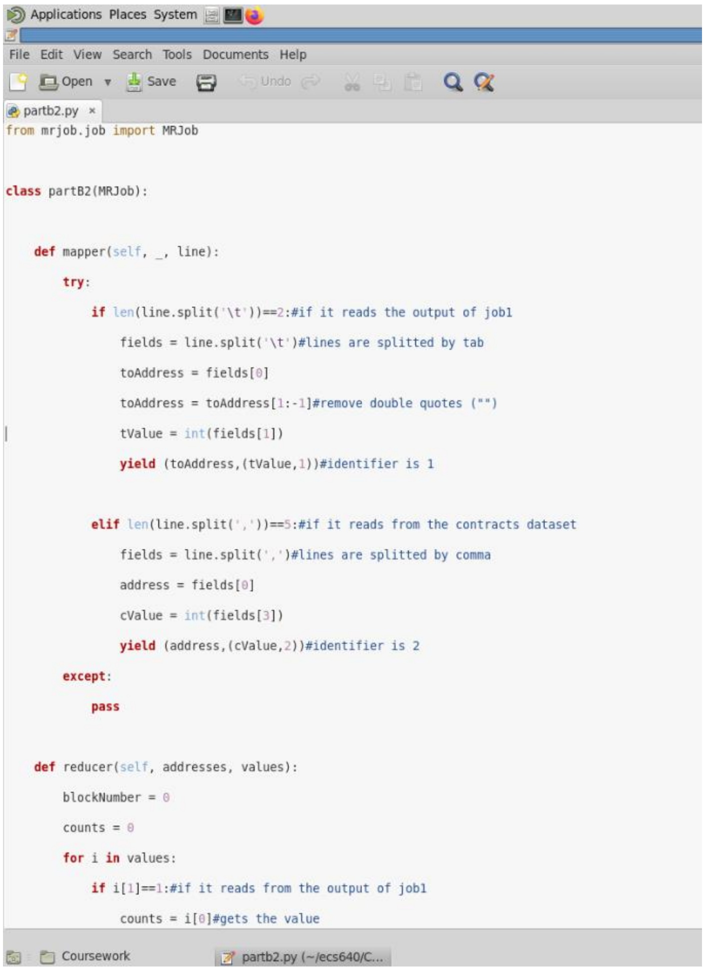
Maps reduce job with key as to_address and value as values from transaction file is used to calculate the initial aggregation. In mapper the lines are split by comma and the key and values are yielded only if the values field is not zero. This will remove all the lines that are not required for computation, thus saving memory and improving the execution performance of the job. A combiner is used to calculate the sum of values for each transaction present in each mapper. Adding a combiner improve the aggregation performance of the job. Finally, in the reducer the same sum operation is used to calculate the aggregate of transaction values. The output is the sum of values in Wei for each transaction. This tells us how much each address within the user space has been involved in. To measure the number of values for each transaction present in each mapper, a combiner is used. The inclusion of a combiner improves the task's aggregation efficiency. Finally, the same sum operation is used in the reducer to compute the aggregate of transaction values. For a transaction, the result is the number of values in Wei. This shows us how much was involved in each address inside the user space.

JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING

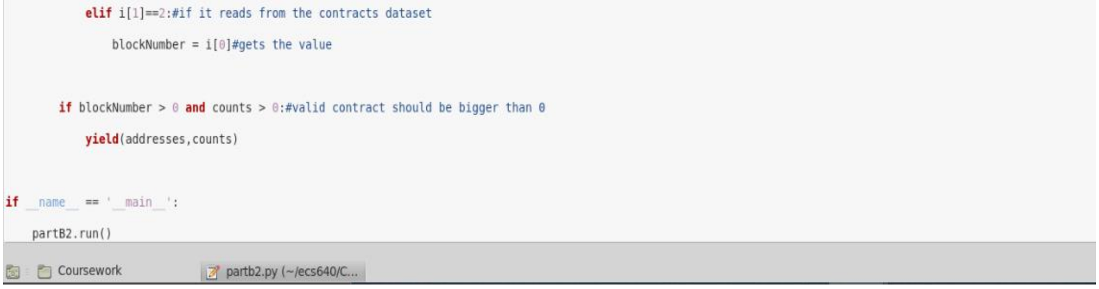
Once you have obtained this aggregate of the transactions, the next step is to perform a repartition join between this aggregate and contracts (example here). You will want to join the to_addressfield from the output of Job 1 with the address field of contracts Secondly, in the reducer, if the address for a given aggregate from Job 1 was not present withincontractsthis should be filtered out as it is a user address and not a smart contract.

Approach: As suggested, to filter out user address from smart contract address, perform repartition join between contracts dataset and transactions aggregate dataset (output from job 1). Only the address that belongs to smart contracts will have the final list of records.

Code: Program name is Partb2.py


Applications Places System
File Edit View Search Tools Documents Help
Open
Save
Undo
partb2.py ×
Coursework
partb2.py (~/ecs640/C...

```
from mrjob.job import MRJob


class partB2(MRJob):


    def mapper(self, _, line):
        try:
            if len(line.split('\t'))==2:#if it reads the output of job1
                fields = line.split('\t')#lines are splitted by tab
                toAddress = fields[0]
                toAddress = toAddress[1:-1]#remove double quotes ("")
                tValue = int(fields[1])
                yield (toAddress,(tValue,1))#identifier is 1

            elif len(line.split(','))==5:#if it reads from the contracts dataset
                fields = line.split(',')#lines are splitted by comma
                address = fields[0]
                cValue = int(fields[3])
                yield (address,(cValue,2))#identifier is 2
        except:
            pass

    def reducer(self, addresses, values):
        blockNumber = 0
        counts = 0
        for i in values:
            if i[1]==1:#if it reads from the output of job1
                counts = i[0]#gets the value
```



```
            elif i[1]==2:#if it reads from the contracts dataset
                blockNumber = i[0]#gets the value

        if blockNumber > 0 and counts > 0:#valid contract should be bigger than 0
            yield(addresses,counts)

if __name__ == '__main__':
    partB2.run()
```

Job id: http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_7288/

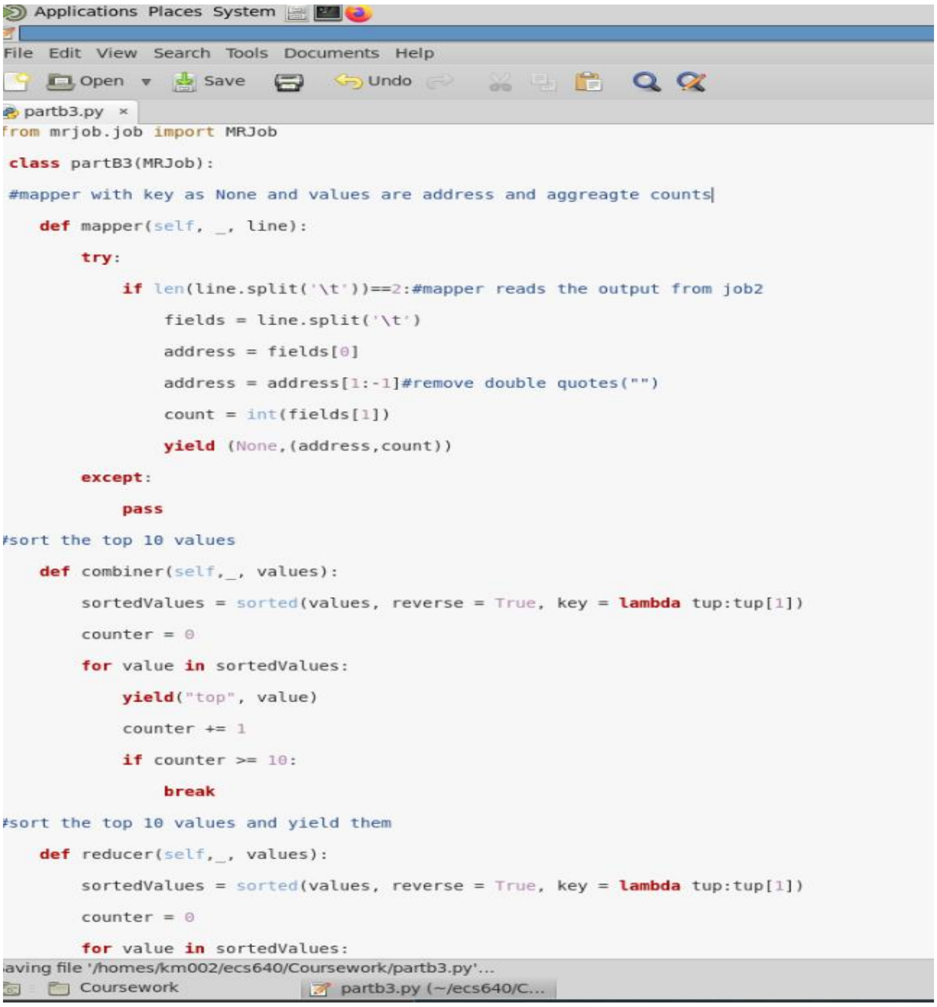
Map reduces repartition work between contract dataset and transaction aggregate dataset (output from job 1). We separate the two input files in the mapper by testing the number of fields in the file. If the number of fields is 5, his dataset of contracts and the number of fields is 2, his combined dataset of transactions is (output from job 1). The first if mapper state recognizes the dataset of contracts in which we define key as address(fields[0]) and value as block number and'1 '. 1 is hardcoded to understand that the value is from the reducer dataset of contracts. The second condition in the mapper understands the aggregate dataset of transactions where we state key as address and value as aggregate count as value (sum of values inWei).

The mapper only gathers records if the two keys fit. That is only if all addresses fit the dataset. I filter the smart contract address in this manner. I find the aggregate values count from the set of values returned by the mapper in the reducer using for loop. If value[1]== 2, the aggregate value is then counted. Eventually, the key that is the smart contract address and value that is the aggregate value is returned in the reducer that gives us the aggregate values of all smart values

JOB 3 -TOP TEN

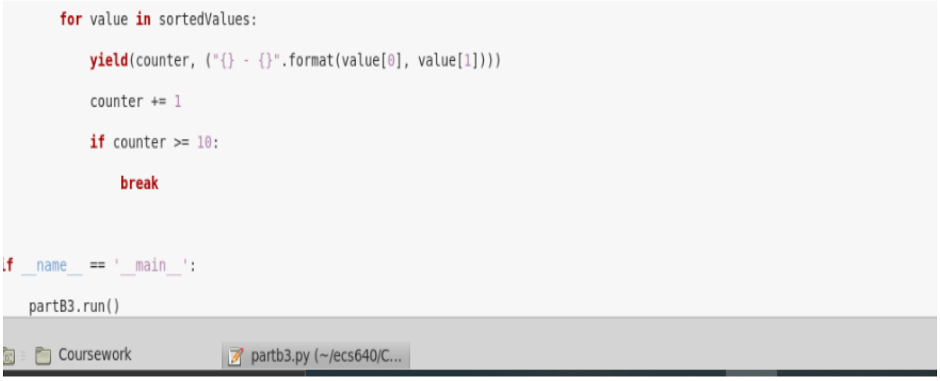
Finally, the third job will take as input the now filtered address aggregates and sort these via a top ten reducer, utilizing what you have learned from lab 4.

Approach: As given, the output from Job 2 will be taken as input and sort the values based on the total aggregate value to get the top 10 values


Applications Places System
File Edit View Search Tools Documents Help
Open
Save
Undo
partb3.py ×
Coursework
partb3.py (~/ecs640/C...

```
from mrjob.job import MRJob
class partB3(MRJob):
#mapper with key as None and values are address and aggreagte counts
    def mapper(self, _, line):
        try:
            if len(line.split('\t'))==2:#mapper reads the output from job2
                fields = line.split('\t')
                address = fields[0]
                address = address[1:-1]#remove double quotes("")
                count = int(fields[1])
                yield (None,(address,count))
        except:
            pass
#sort the top 10 values
    def combiner(self,_, values):
        sortedValues = sorted(values, reverse = True, key = lambda tup:tup[1])
        counter = 0
        for value in sortedValues:
            yield("top", value)
            counter += 1
            if counter >= 10:
                break
#sort the top 10 values and yield them
    def reducer(self,_, values):
        sortedValues = sorted(values, reverse = True, key = lambda tup:tup[1])
        counter = 0
        for value in sortedValues:
```

aving file '/homes/km002/ecs640/Coursework/partb3.py'...



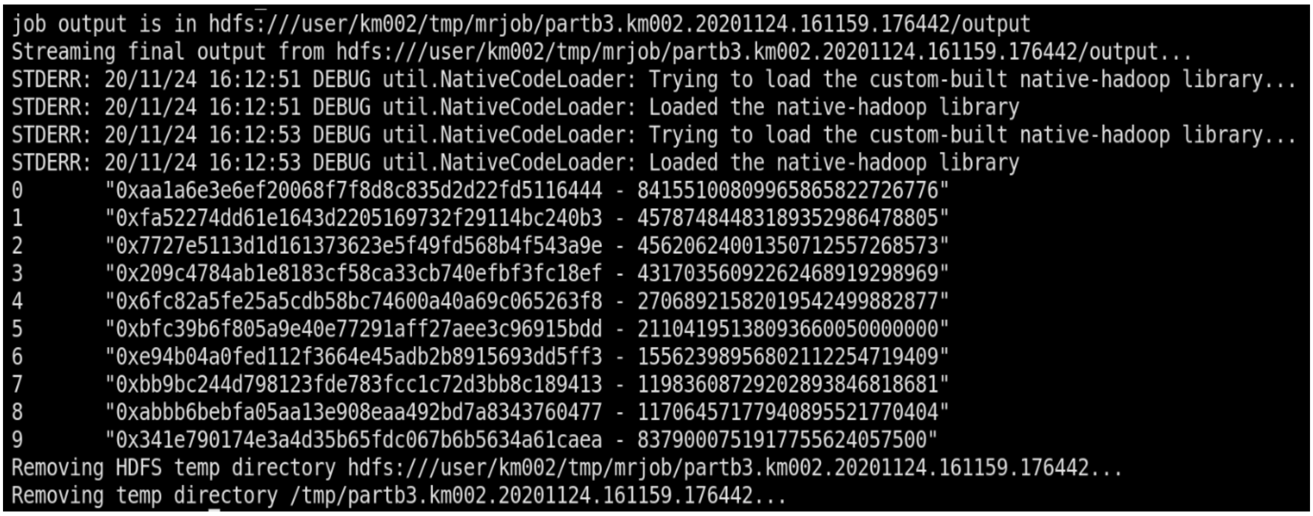
```
        for value in sortedValues:
            yield(counter, ("{} - {}".format(value[0], value[1])))
            counter += 1
            if counter >= 10:
                break

if __name__ == '__main__':
    partB3.run()
```

Job id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_7325/

Map decreases function to filter out the top 10 most common resources received as a result of output from Job 2. In the mapper, when the input file is divided by tab, the lines are broken by tab. The key is zero, and the address and sum of the combined values is the value. Main is none because we need to sort the values depending on the sum count. In order to accelerate the sorting process, a combiner is used. Sort the values in ascending order by giving reverse = True in the combiner. This will first return the top values. A for loop is used to iterate between all records after this condition has been set and make the sort values in ascending order. Finally, the same combiner operation is done in the reducer, where key value pairs are sorted again from all the combiners to achieve the same result. To iterate between all records from the combiner, A for loop is used and the result is shown in the top 10 order. The for loop is terminated until the iteration count approaches 11 to show just the top 10 values. Output is the top 10 most common services.

Below are top 10 services (output from Job3),



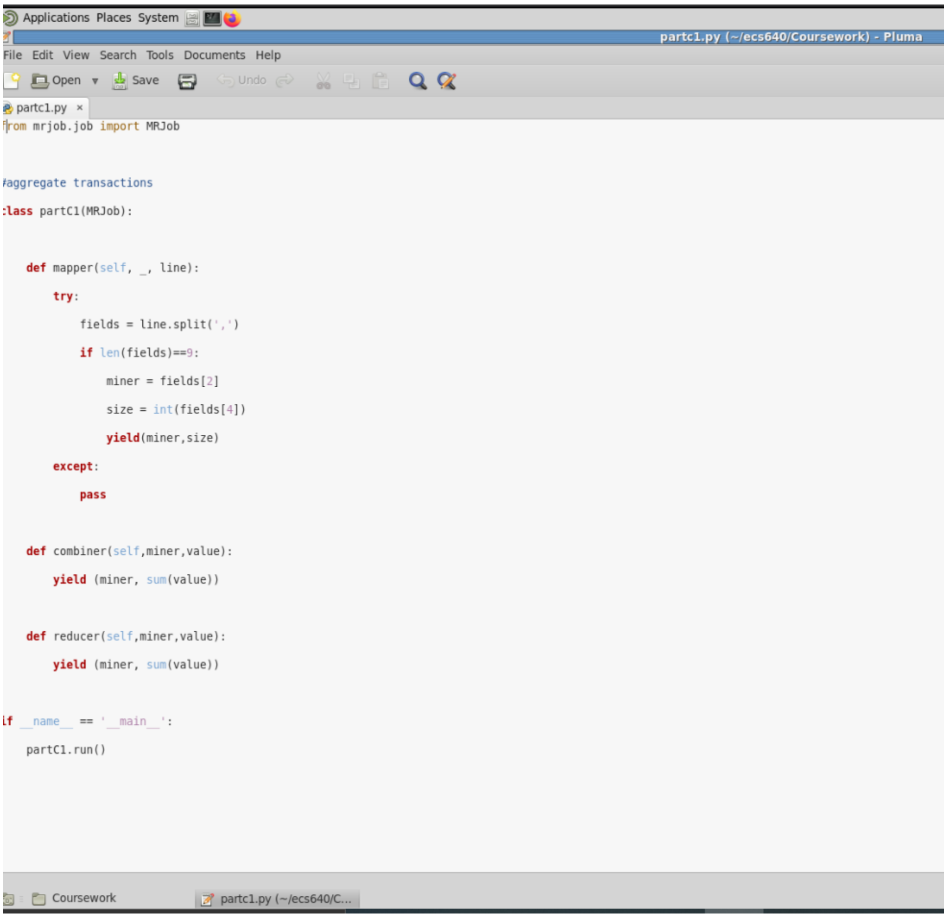
```
job output is in hdfs:///user/km002/tmp/mrjob/partb3.km002.20201124.161159.176442/output
Streaming final output from hdfs:///user/km002/tmp/mrjob/partb3.km002.20201124.161159.176442/output...
STDERR: 20/11/24 16:12:51 DEBUG util.NativeCodeLoader: Trying to load the custom-built native-hadoop library...
STDERR: 20/11/24 16:12:51 DEBUG util.NativeCodeLoader: Loaded the native-hadoop library
STDERR: 20/11/24 16:12:53 DEBUG util.NativeCodeLoader: Trying to load the custom-built native-hadoop library...
STDERR: 20/11/24 16:12:53 DEBUG util.NativeCodeLoader: Loaded the native-hadoop library
0       "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 - 84155100809965865822726776"
1       "0xfa52274dd61e1643d2205169732f29114bc240b3 - 45787484483189352986478805"
2       "0x7727e5113d1d161373623e5f49fd568b4f543a9e - 45620624001350712557268573"
3       "0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef - 43170356092262468919298969"
4       "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8 - 27068921582019542499882877"
5       "0xbfc39b6f805a9e40e77291aff27aee3c96915bdd - 21104195138093660050000000"
6       "0xe94b04a0fed112f3664e45adb2b8915693dd5ff3 - 15562398956802112254719409"
7       "0xbb9bc244d798123fde783fcc1c72d3bb8c189413 - 11983608729202893846818681"
8       "0xabbb6bebfa05aa13e908eaa492bd7a8343760477 - 11706457177940895521770404"
9       "0x341e790174e3a4d35b65fdc067b6b5634a61caea - 8379000751917755624057500"
Removing HDFS temp directory hdfs:///user/km002/tmp/mrjob/partb3.km002.20201124.161159.176442...
Removing temp directory /tmp/partb3.km002.20201124.161159.176442...
```

PART C: TOP TEN MOST ACTIVE MINERS

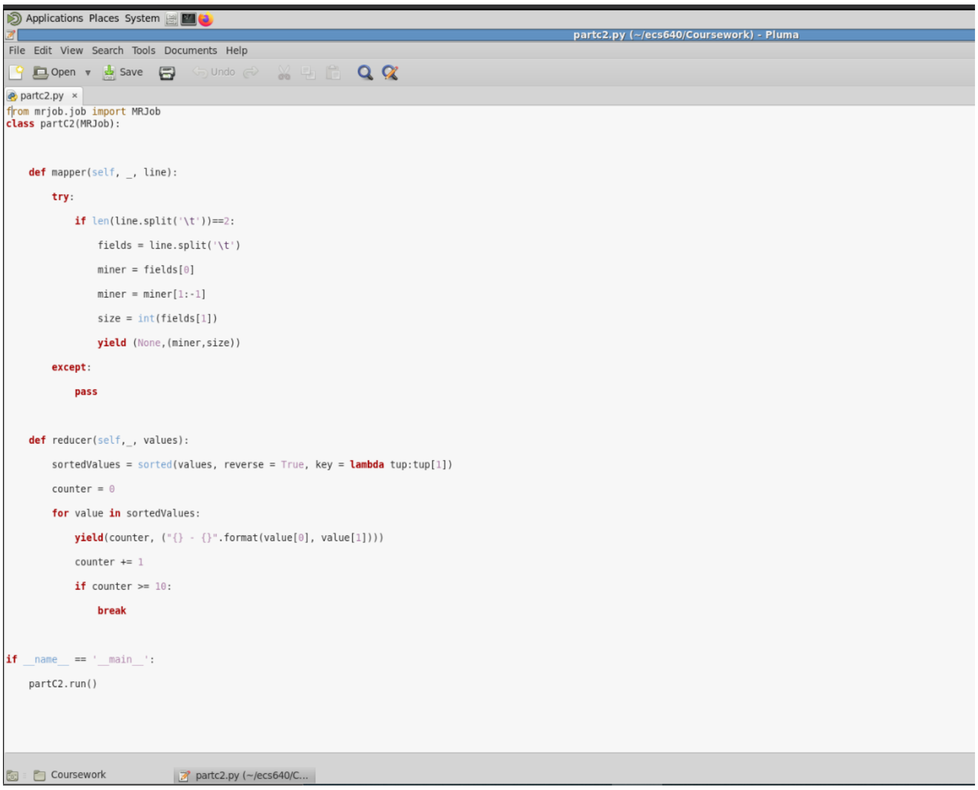
Approach: As given, the value field is aggregated for addresses in the to address field from the transaction file to determine the aggregation of transactions.

Code: Hadoop Map/Reduce program to calculate the initial aggregation is given below. Program name is Partc1.py.

Job id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_7384/


Applications Places System
partc1.py (~/ecs640/Coursework) - Pluma
File Edit View Search Tools Documents Help
Open
Save
Undo
partc1.py
Coursework
partc1.py (~/ecs640/C...

```
from mrjob.job import MRJob


#aggregate transactions

class partC1(MRJob):


    def mapper(self, _, line):

        try:

            fields = line.split(',')

            if len(fields)==9:

                miner = fields[2]

                size = int(fields[4])

                yield(miner,size)

        except:

            pass


    def combiner(self,miner,value):

        yield (miner, sum(value))


    def reducer(self,miner,value):

        yield (miner, sum(value))


if __name__ == '__main__':

    partC1.run()
```

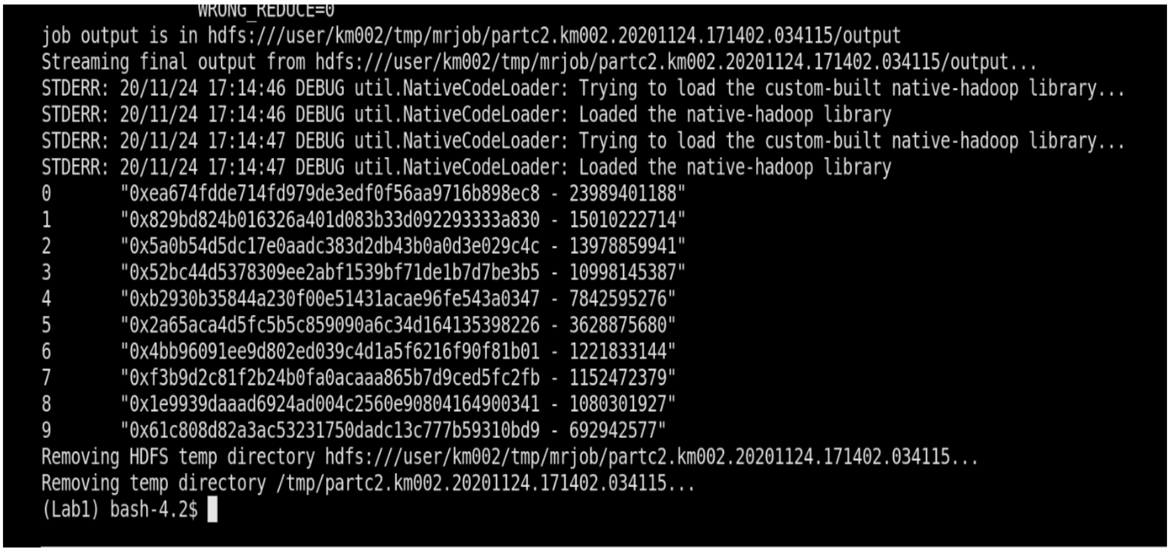

Applications Places System
partc2.py (~/ecs640/Coursework) - Pluma
File Edit View Search Tools Documents Help
Open
Save
Undo
partc2.py
Coursework
partc2.py (~/ecs640/C...

```
from mrjob.job import MRJob
class partC2(MRJob):


    def mapper(self, _, line):

        try:

            if len(line.split('\t'))==2:

                fields = line.split('\t')

                miner = fields[0]

                miner = miner[1:-1]

                size = int(fields[1])

                yield (None,(miner,size))

        except:

            pass


    def reducer(self,_, values):

        sortedValues = sorted(values, reverse = True, key = lambda tup:tup[1])

        counter = 0

        for value in sortedValues:

            yield(counter, ("{} - {}".format(value[0], value[1])))

            counter += 1

            if counter >= 10:

                break


if __name__ == '__main__':

    partC2.run()
```

Map reduces work to screen out the top 10 most productive miners out of job 1 production. The number of characters in a file should be 2. The lines are divided by a tab, since the input file is separated by a tab. Yield Zero as the key and address as values and the sum of the aggregated count. Since we need to filter the values based on the aggregated count, the key is None.

By defining reverse = True, the combiner is used to arrange the values in decreasing order. This will first supply us with the top values. A for loop is used to iterate the sorted values across and generate in decreasing order the first top ten. With the combiner, the reducer does the same operation. Finally, once the iteration count exceeds 10 and the reducer reveals the top 10 values, the loop is terminated.

Job id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1604349877093_7402/



```
                WRONG_REDUCE=0
job output is in hdfs:///user/km002/tmp/mrjob/partc2.km002.20201124.171402.034115/output
Streaming final output from hdfs:///user/km002/tmp/mrjob/partc2.km002.20201124.171402.034115/output...
STDERR: 20/11/24 17:14:46 DEBUG util.NativeCodeLoader: Trying to load the custom-built native-hadoop library...
STDERR: 20/11/24 17:14:46 DEBUG util.NativeCodeLoader: Loaded the native-hadoop library
STDERR: 20/11/24 17:14:47 DEBUG util.NativeCodeLoader: Trying to load the custom-built native-hadoop library...
STDERR: 20/11/24 17:14:47 DEBUG util.NativeCodeLoader: Loaded the native-hadoop library
0       "0xea674fdde714fd979de3edf0f56aa9716b898ec8 - 23989401188"
1       "0x829bd824b016326a401d083b33d092293333a830 - 15010222714"
2       "0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c - 13978859941"
3       "0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5 - 10998145387"
4       "0xb2930b35844a230f00e51431acae96fe543a0347 - 7842595276"
5       "0x2a65aca4d5fc5b5c859090a6c34d164135398226 - 3628875680"
6       "0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01 - 1221833144"
7       "0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb - 1152472379"
8       "0x1e9939daaad6924ad004c2560e90804164900341 - 1080301927"
9       "0x61c808d82a3ac53231750dadc13c777b59310bd9 - 692942577"
Removing HDFS temp directory hdfs:///user/km002/tmp/mrjob/partc2.km002.20201124.171402.034115...
Removing temp directory /tmp/partc2.km002.20201124.171402.034115...
(Lab1) bash-4.2$ 
```

Part D. Data Exploration

SCAM ANALYSIS

Part A: Most lucrative form of scam

Approach:

I have entered the scams dataset and transactions dataset based on the transaction key address to recognize the most lucrative type of scam. From the joined dataset the sum of values with scam group as key is determined. The scam group with the highest values is the most profitable type of scam from the combined production.

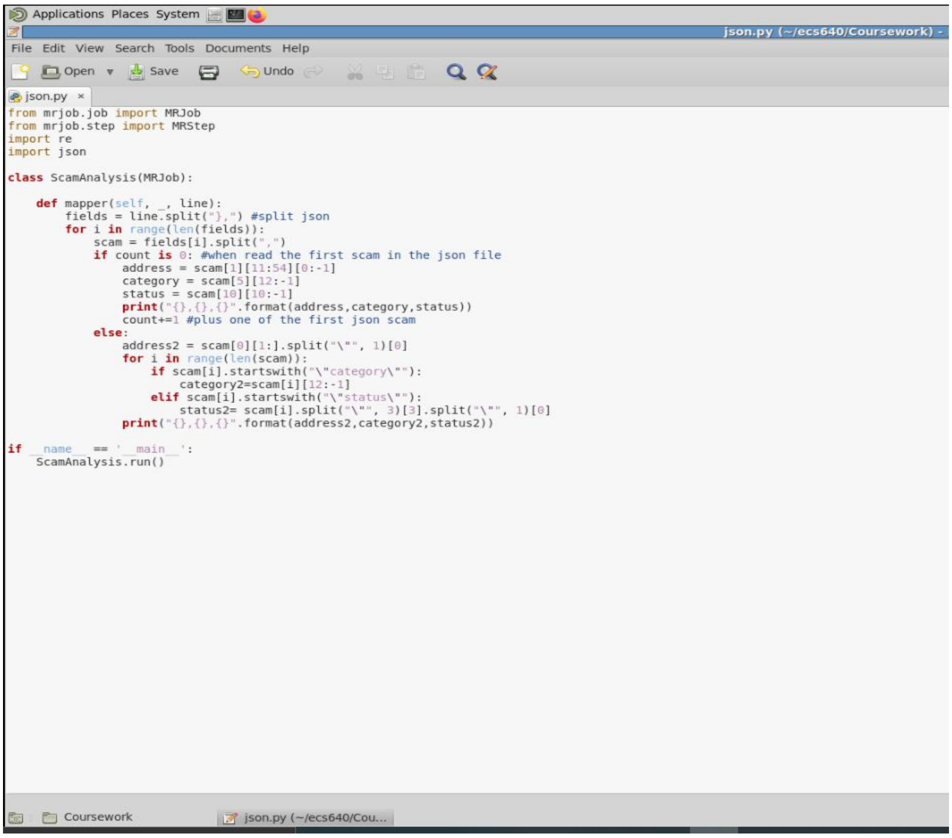
Data Preprocessing:

To access the fields in scam dataset first I converted the scams.json to CSV file by using the below code in json.py

Code: Program names is ScamAnalysis.py

Job id: (The job id has a different username due to an error in my ITL account)

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_8967/

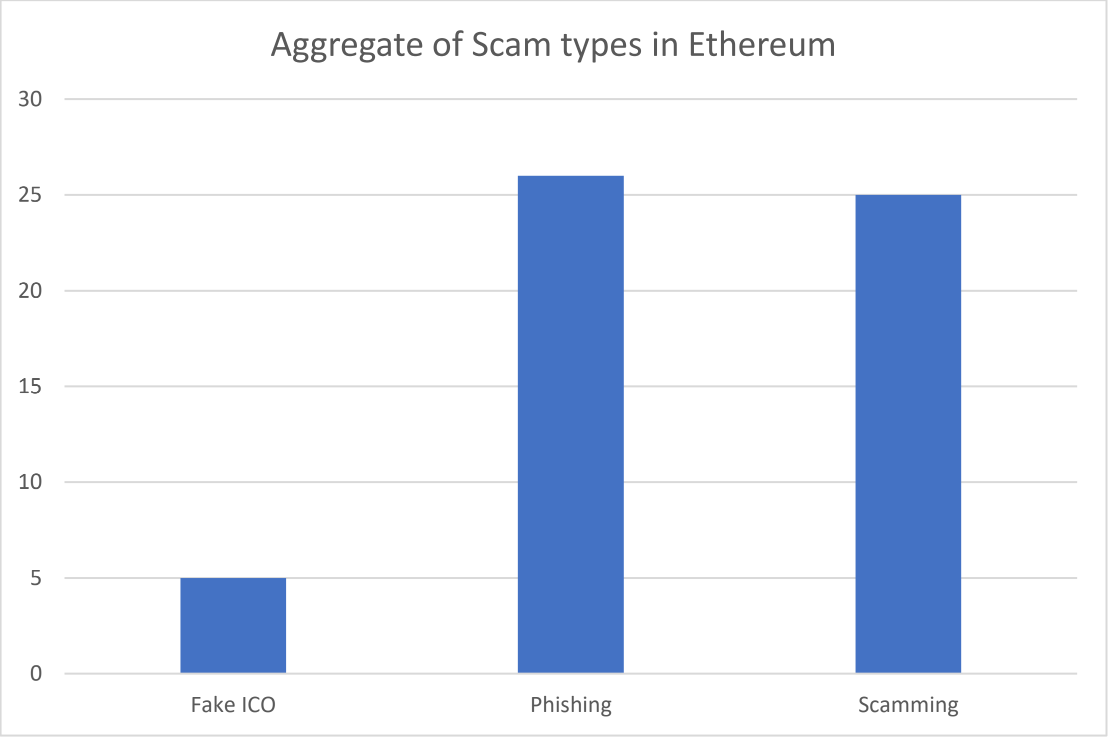

Applications Places System
json.py (~/ecs640/Coursework) -
File Edit View Search Tools Documents Help
Open
Save
Undo
json.py
Coursework
json.py (~/ecs640/Cou...

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import re
import json

class ScamAnalysis(MRJob):

    def mapper(self, _, line):
        fields = line.split("},") #split json
        for i in range(len(fields)):
            scam = fields[i].split(",")
            if count is 0: #when read the first scam in the json file
                address = scam[1][11:54][0:-1]
                category = scam[5][12:-1]
                status = scam[10][10:-1]
                print("{},{},{}".format(address,category,status))
                count+=1 #plus one of the first json scam
            else:
                address2 = scam[0][1:].split("\"", 1)[0]
                for i in range(len(scam)):
                    if scam[i].startswith("\"category\""):
                        category2=scam[i][12:-1]
                    elif scam[i].startswith("\"status\""):
                        status2= scam[i].split("\"", 3)[3].split("\"", 1)[0]
                print("{},{},{}".format(address2,category2,status2))

if __name__ == '__main__':
    ScamAnalysis.run()
```

Result Analysis:

Below is the output,

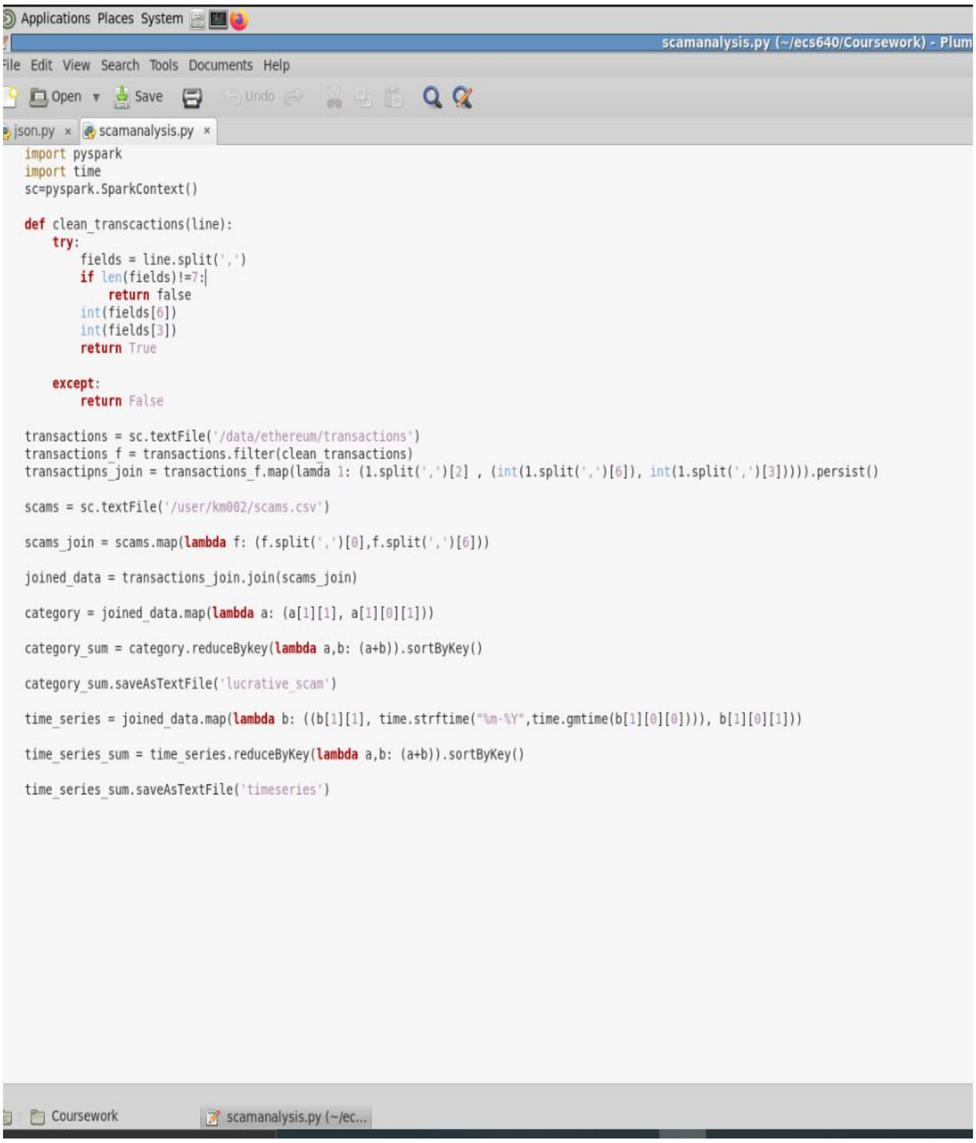
From the result the most lucrative form of scam is “Scamming” as the values transferred in Wei is the highest for this category of scam. The corresponding plot in Ethereum shown below.



Part B: Timeseries analysis of different types of scams

Approach:

Using the joined dataset from previous section the scam category, timestamp values and aggregate of values in wei is used to analyses the trend of different types of scams.


Applications Places System
scamanalysis.py (~/ecs640/Coursework) - Plum
File Edit View Search Tools Documents Help
Open
Save
Undo
json.py
scamanalysis.py
Coursework
scamanalysis.py (~/ec...

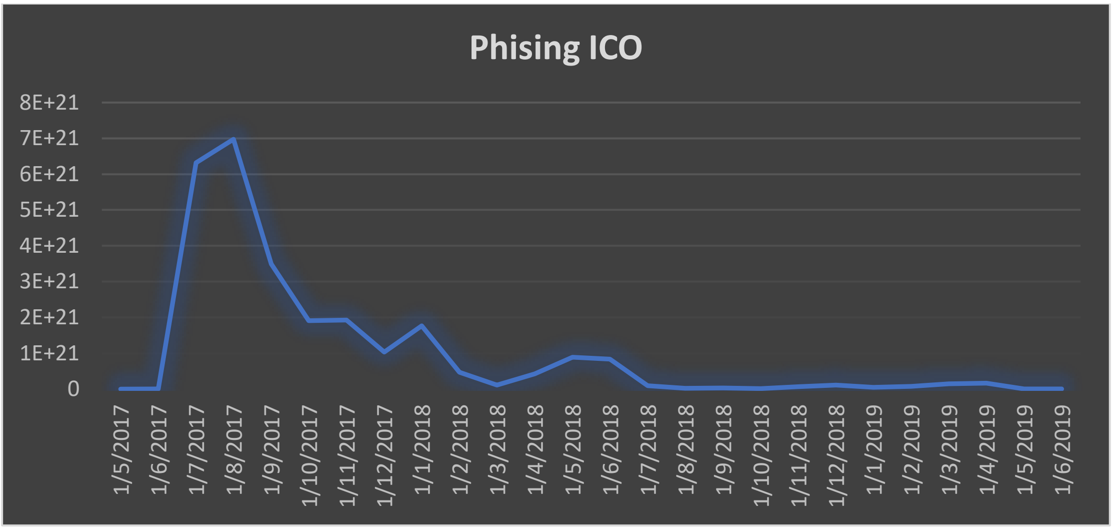
```
import pyspark
import time
sc=pyspark.SparkContext()

def clean_transcactions(line):
    try:
        fields = line.split(',')
        if len(fields)!=7:
            return false
        int(fields[6])
        int(fields[3])
        return True

    except:
        return False

transactions = sc.textFile('/data/ethereum/transactions')
transactions_f = transactions.filter(clean_transactions)
transactipns_join = transactions_f.map(lamda 1: (1.split(',')[2] , (int(1.split(',')[6]), int(1.split(',')[3])))).persist()

scams = sc.textFile('/user/km002/scams.csv')

scams_join = scams.map(lambda f: (f.split(',')[0],f.split(',')[6]))

joined_data = transactions_join.join(scams_join)

category = joined_data.map(lambda a: (a[1][1], a[1][0][1]))

category_sum = category.reduceBykey(lambda a,b: (a+b)).sortByKey()

category_sum.saveAsTextFile('lucrative_scam')

time_series = joined_data.map(lambda b: ((b[1][1], time.strftime("%m-%Y",time.gmtime(b[1][0][0]))), b[1][0][1]))

time_series_sum = time_series.reduceByKey(lambda a,b: (a+b)).sortByKey()

time_series_sum.saveAsTextFile('timeseries')
```

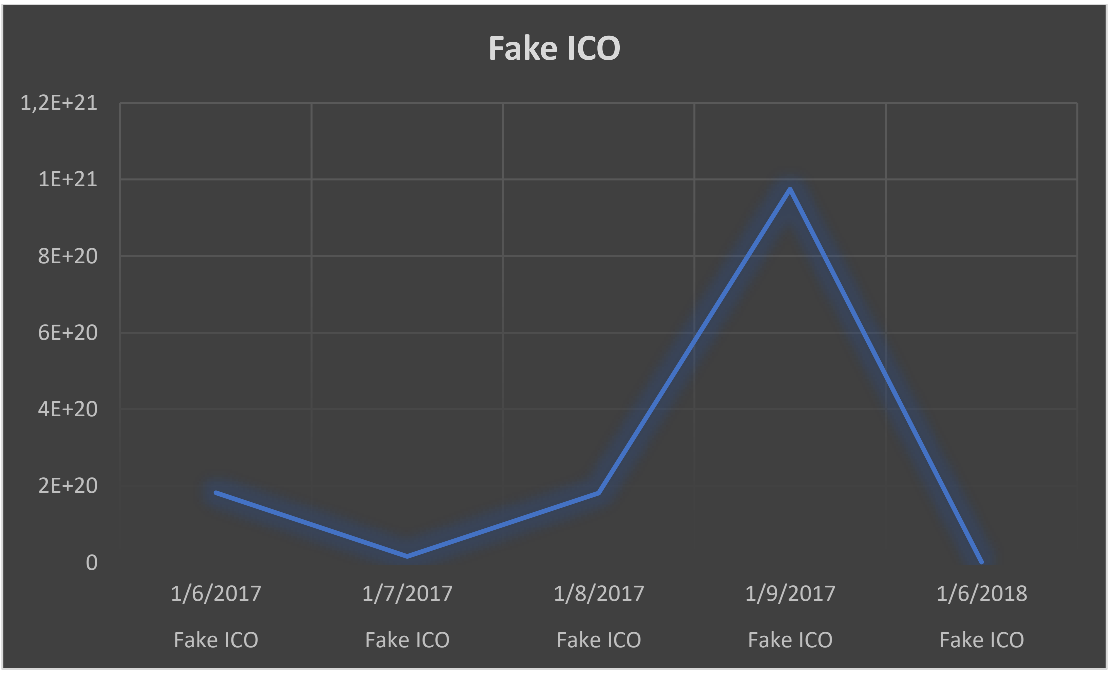
Code Explanation:

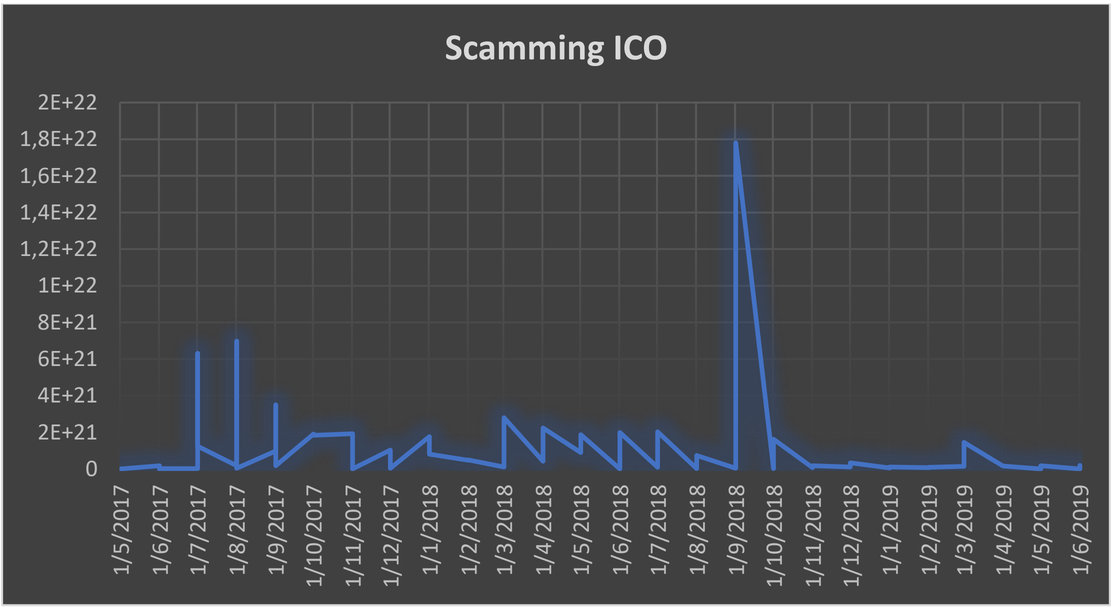
From the joined dataset spark's map function is used tomap the key as scam category and timestamp and value as values in wei. The Unix timestamp is converted to Gregorian format using time function. Spark's reduce by key method is used to calculate the aggregate of values in wei. The result of the function is the scam category and its time period along with sum of values in wei for each month. The result is finally stored in HDFS as timeseries.

Result Analysis:

Below are the plots of different types of scams that changes over time,


Phising ICO
8E+21
7E+21
6E+21
5E+21
4E+21
3E+21
2E+21
1E+21
0
1/5/2017
1/6/2017
1/7/2017
1/8/2017
1/9/2017
1/10/2017
1/11/2017
1/12/2017
1/1/2018
1/2/2018
1/3/2018
1/4/2018
1/5/2018
1/6/2018
1/7/2018
1/8/2018
1/9/2018
1/10/2018
1/11/2018
1/12/2018
1/1/2019
1/2/2019
1/3/2019
1/4/2019
1/5/2019
1/6/2019


Fake ICO
1,2E+21
1E+21
8E+20
6E+20
4E+20
2E+20
0
1/6/2017
1/7/2017
1/8/2017
1/9/2017
1/6/2018
Fake ICO
Fake ICO
Fake ICO
Fake ICO
Fake ICO


Scamming ICO
2E+22
1,8E+22
1,6E+22
1,4E+22
1,2E+22
1E+22
8E+21
6E+21
4E+21
2E+21
0
1/5/2017
1/6/2017
1/7/2017
1/8/2017
1/9/2017
1/10/2017
1/11/2017
1/12/2017
1/1/2018
1/2/2018
1/3/2018
1/4/2018
1/5/2018
1/6/2018
1/7/2018
1/8/2018
1/9/2018
1/10/2018
1/11/2018
1/12/2018
1/1/2019
1/2/2019
1/3/2019
1/4/2019
1/5/2019
1/6/2019

All the values of the Wei are transformed and plotted into ether. We can see from the graph that "Scamming" was the most common scam at its height during Sep-2018.

Gas Guzzlers

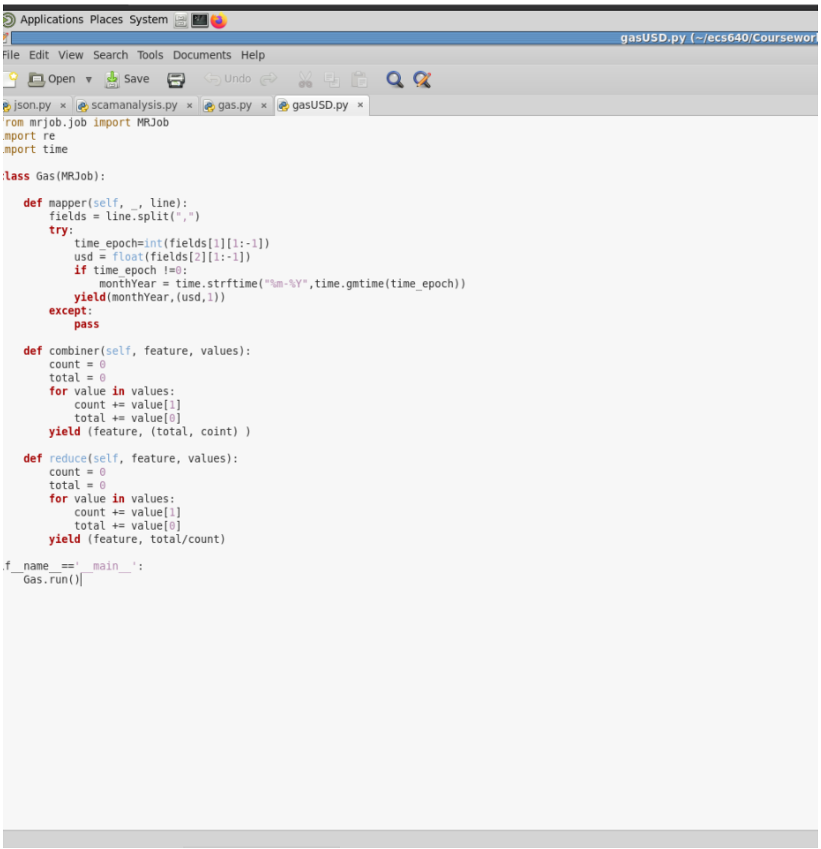
Part 1: Time series analysis of gas price change over the years(in Wei).

Approach:

In order to quantify the gas price shift over the years, I have used the fields gas price and block timestamp values from the transaction dataset. The time series study of how the price has risen over the years is given by the average amount of the gas price per month.

Code:

The program is gasUSD.


Applications Places System
gasUSD.py (~/ecs640/Coursewor
File Edit View Search Tools Documents Help
Open
Save
Undo
json.py
scamanalysis.py
gas.py
gasUSD.py

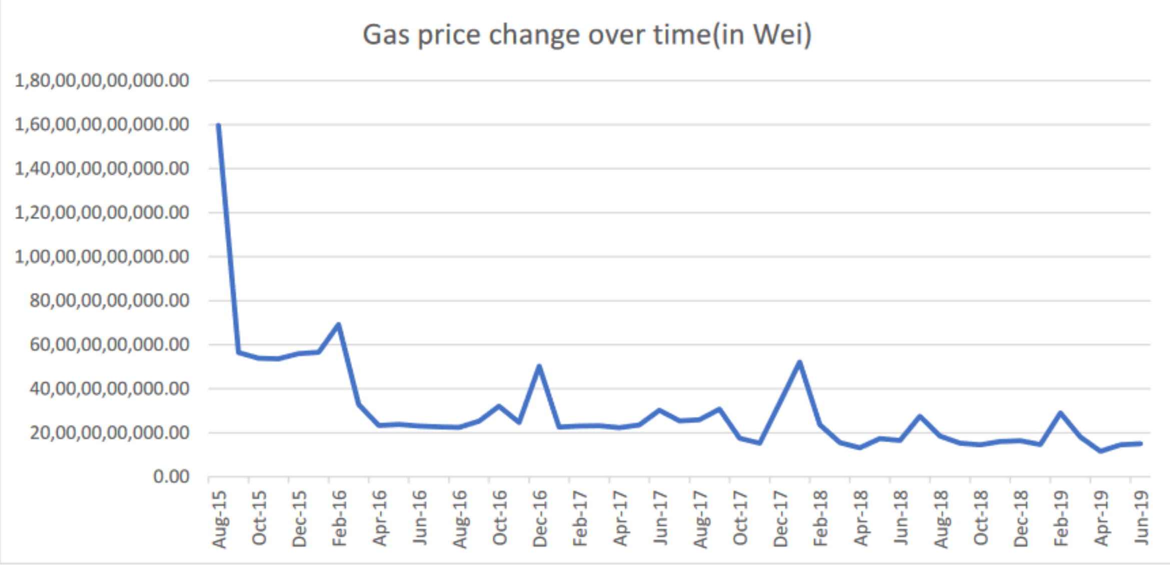
```
rom mrjob.job import MRJob
mport re
mport time

lass Gas(MRJob):

   def mapper(self, _, line):
       fields = line.split(",")
       try:
           time_epoch=int(fields[1][1:-1])
           usd = float(fields[2][1:-1])
           if time_epoch !=0:
               monthYear = time.strftime("%m-%Y",time.gmtime(time_epoch))
           yield(monthYear,(usd,1))
       except:
           pass

   def combiner(self, feature, values):
       count = 0
       total = 0
       for value in values:
           count += value[1]
           total += value[0]
       yield (feature, (total, coint) )

   def reduce(self, feature, values):
       count = 0
       total = 0
       for value in values:
           count += value[1]
           total += value[0]
       yield (feature, total/count)

f __name__=='__main__':
   Gas.run()
```

Job id: (The job id has a different username due to an error in my ITL account)

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_8689

Code Explanation:

To analyses the gas price change over years the gas price value in Wei and timestamp is used to perform the time series analysis. The average of gas price value per month gives us an insight of the gas price change over years. A map reduce job is used to perform this computation. In mapper key is block_timestamp field and values are gas_price and count 1 to count the number of occurrences which is used later in reducer to calculate the average. The Unix timestamp is converted to Gregorian format using time function. In combiner the key is the formatted timestamp and values are gas price and count. A for loop is used to calculate the sum of gas values and its number of occurrences per month. In reducer the total the same for loop logic is implemented and the yielded values are formatted timestamp and average of gas price per month. The average is calculated by dividing the sum of gas price with its total number of occurrences. Finally, the result is the average of gas price per month.



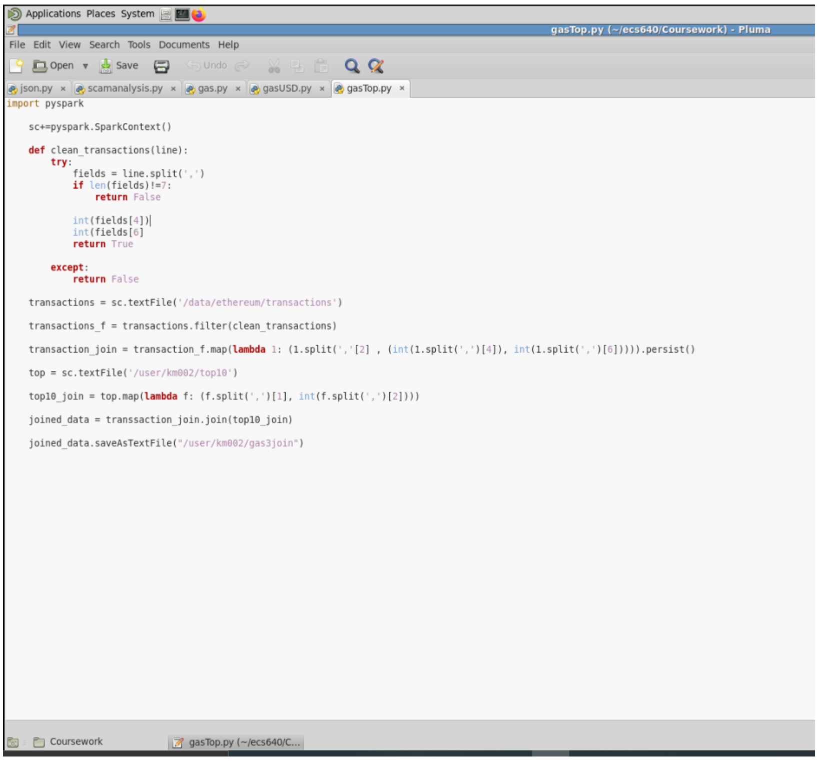
Result Analysis:

The obtained results from the map reduce job is plotted using excel and below is the time series analysis graph of gas price change over years. The plot shows that the price was more during the inception of Ethereum in Aug 2015 and there is a sudden decrease in value next month. The gas price value appears to be stable after September 2015, with minor variations until Feb 2016. The price of gas decreases gradually after Feb 2016 and almost seems to be stable with small fluctuations until Jun 2019.

Part 1: Time series analysis of Gas consumed by top 10 smart contracts over the years.

Approach: A replication joint is carried out between the top 10 smart contracts and transaction datasets in order to measure the volume of gas consumed by the top 10 contracts over the years. Replication join's output is top 10 addresses, its gas supplied by sender values. The average gas consumed over the years per address is measured after joining the operation.

Code: Program name is gastop.py


Applications Places System
gasTop.py (~/ecs640/Coursework) - Pluma
File Edit View Search Tools Documents Help
Open
Save
Undo
json.py
scamanalysis.py
gas.py
gasUSD.py
gasTop.py
Coursework
gasTop.py (~/ecs640/C...

```
import pyspark

    sc+=pyspark.SparkContext()

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

            int(fields[4])
            int(fields[6]
            return True

        except:
            return False

    transactions = sc.textFile('/data/ethereum/transactions')

    transactions_f = transactions.filter(clean_transactions)

    transaction_join = transaction_f.map(lambda 1: (1.split(','[2] , (int(1.split(',')[4]), int(1.split(',')[6])))).persist()

    top = sc.textFile('/user/km002/top10')

    top10_join = top.map(lambda f: (f.split(',')[1], int(f.split(',')[2])))

    joined_data = transsaction_join.join(top10_join)

    joined_data.saveAsTextFile("/user/km002/gas3join")
```

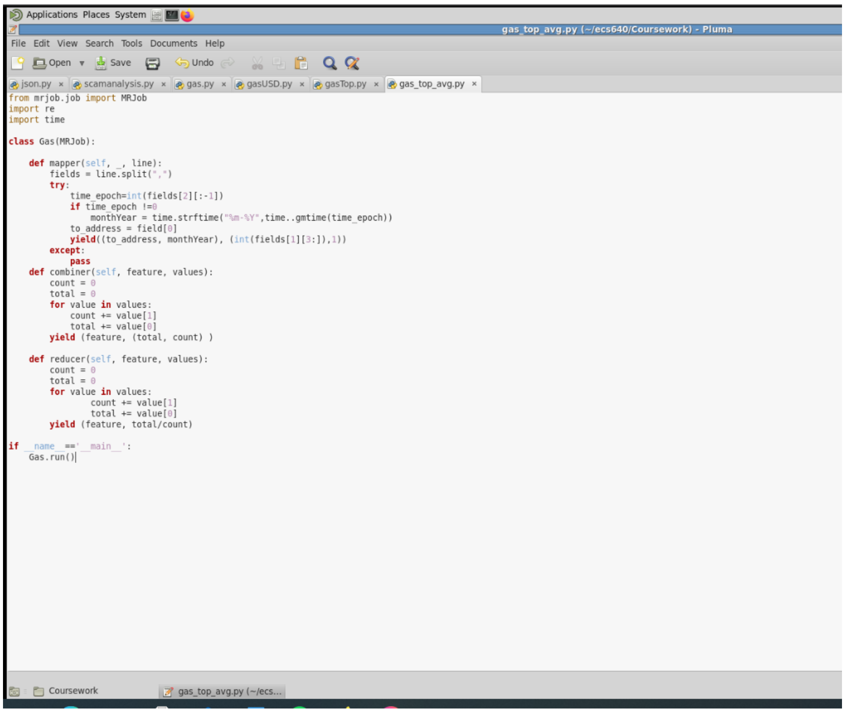
Job id: (The job id has a different username due to an error in my ITL account)

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_8689

Code Explanation:

Spark operates on the top 10 smart contract datasets and activity datasets to enter activities. The data collection of transactions is read, and the map operation is done. The key is to answer, and gas and timestamp field values are values. The product of the map is stored in the vector Transactions join. The Top 10 dataset of smart contracts is read, and operation of the map is carried out. The key is address and aggregate counts are value. The map result is stored in the vector top10 join. Using join operation, the information is joined, and the key is address and gas, timestamp and aggregate value are values. The product of join operation is stored in the HDFS called gastopjoin

Step 2 – Calculate the average of gas consumed by top 10 smart contracts.



Applications Places System
gas_top_avg.py (~/ecs640/Coursework) - Pluma
File Edit View Search Tools Documents Help
Open
Save
Undo
json.py
scamanalysis.py
gas.py
gasUSD.py
gasTop.py
gas_top_avg.py
Coursework
gas_top_avg.py (~/ecs...

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

class Gas(MRJob):

    def mapper(self, _, line):
        fields = line.split(",")
        try:
            time_epoch=int(fields[2][:-1])
            if time_epoch !=0
                monthYear = time.strftime("%m-%Y",time..gmtime(time_epoch))
            to_address = field[0]
            yield((to_address, monthYear), (int(fields[1][3:]),1))
        except:
            pass
    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total, count) )

    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
                count += value[1]
                total += value[0]
        yield (feature, total/count)

if __name__=='__main__':
    Gas.run()
```

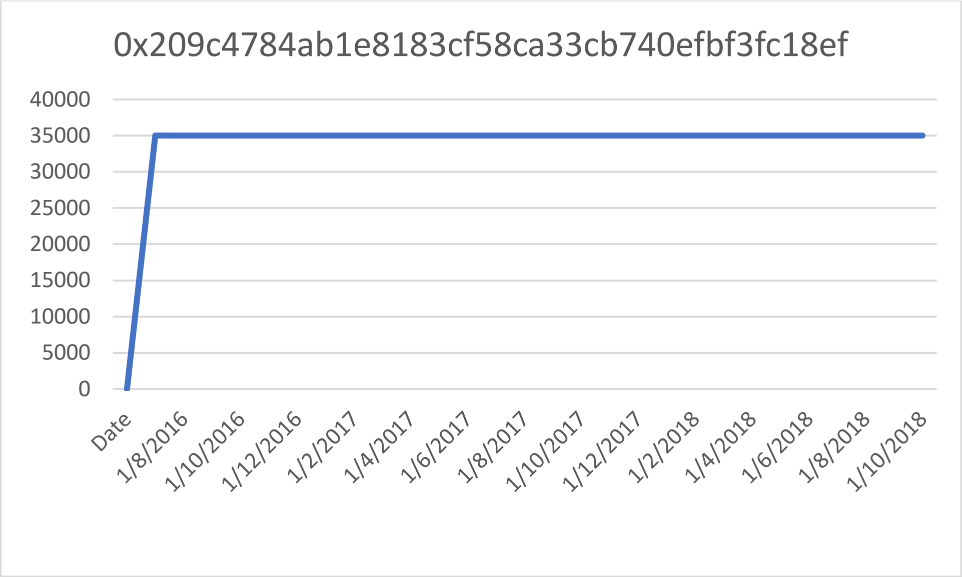
Code: Program name is gastop_avg.py

Code Explanation:

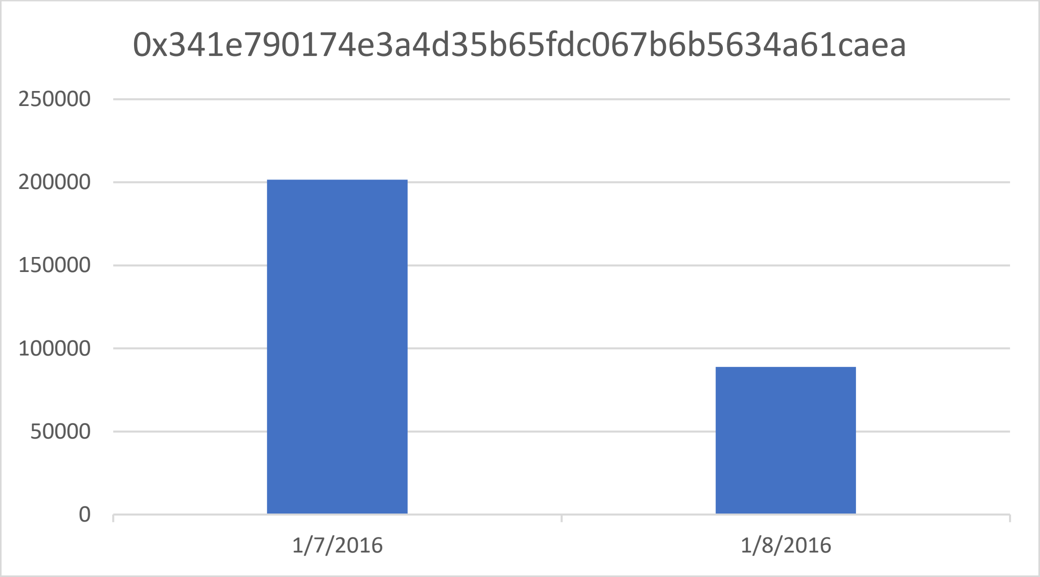
Map reduce job to calculate the average of gas consumed per smart contract from its start time to end time. Mapper is defined with key as to_address and formatted timestamp. The Unix timestamp is converted to Gregorian format using time library. The values of the mapper are gas provided by sender along with count 1. The input dataset is gastopjoin obtained from previous step. Combiner is defined with key as to_address and formatted timestampand values are sum of gas and the number of occurrences in each address. Reducer is defined with key as to_address and formatted timestamp. Values are average of gas consumedper smart contract from its start time to end time.

Result Analysis:

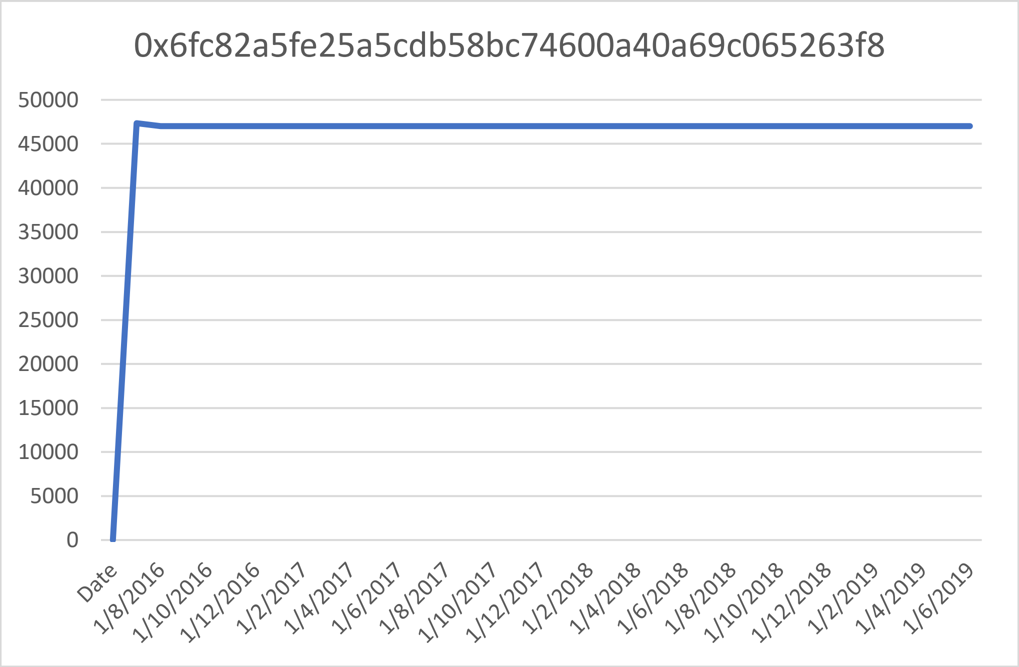
The end result is the average gas absorbed from its start time to the end time by the top 10 smart contracts every month. Below is a set of graphs plotted with the results.



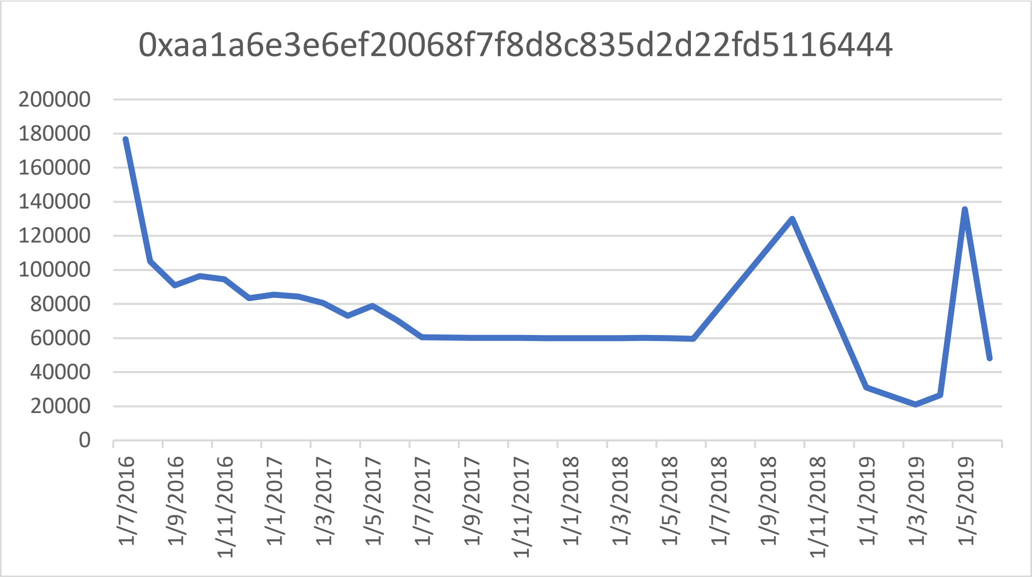
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef
40000
35000
30000
25000
20000
15000
10000
5000
0
Date
1/8/2016
1/10/2016
1/12/2016
1/2/2017
1/4/2017
1/6/2017
1/8/2017
1/10/2017
1/12/2017
1/2/2018
1/4/2018
1/6/2018
1/8/2018
1/10/2018

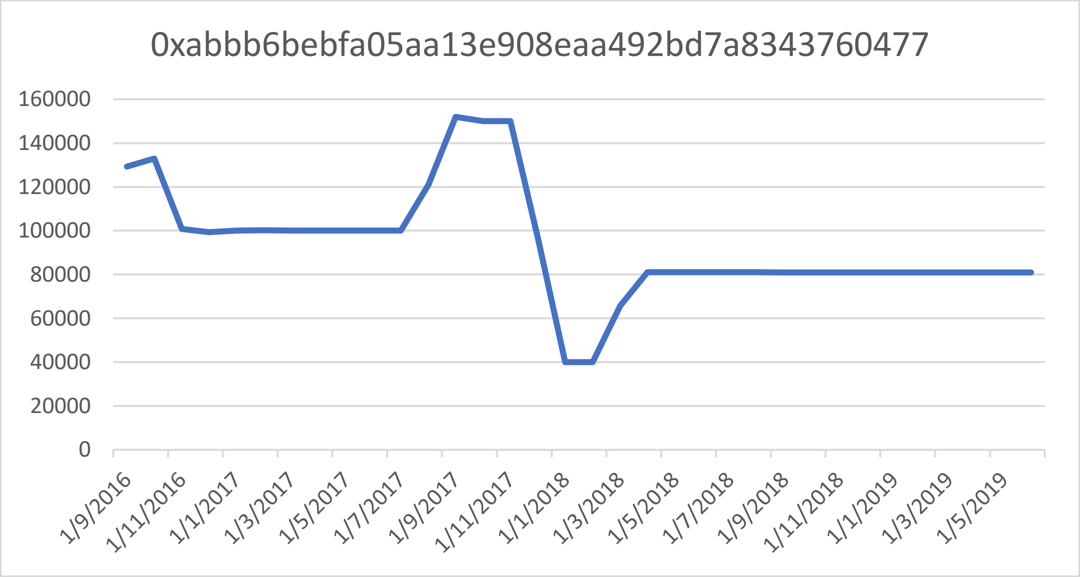


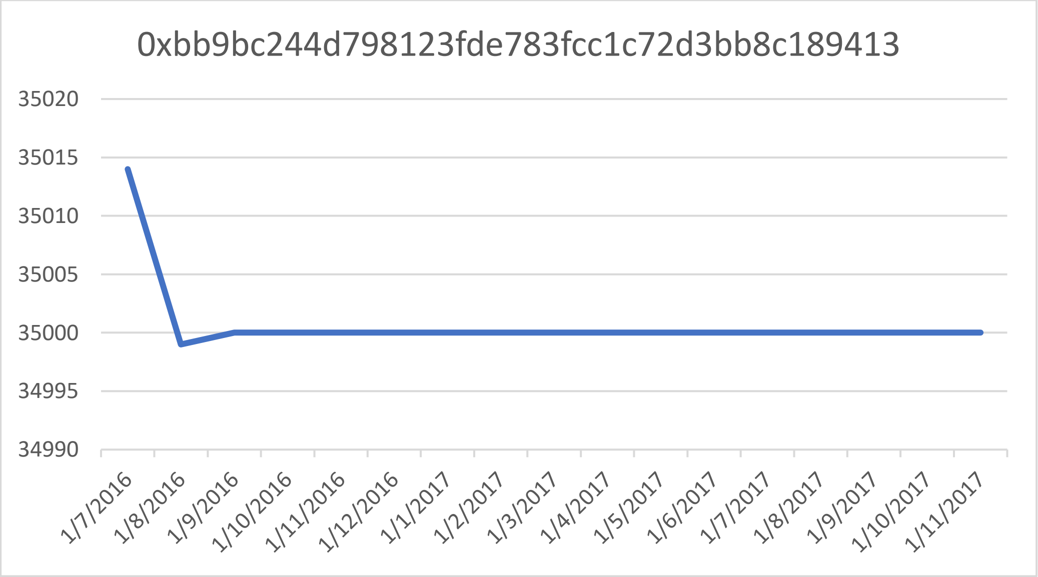
0x341e790174e3a4d35b65fdc067b6b5634a61caea
250000
200000
150000
100000
50000
0
1/7/2016
1/8/2016

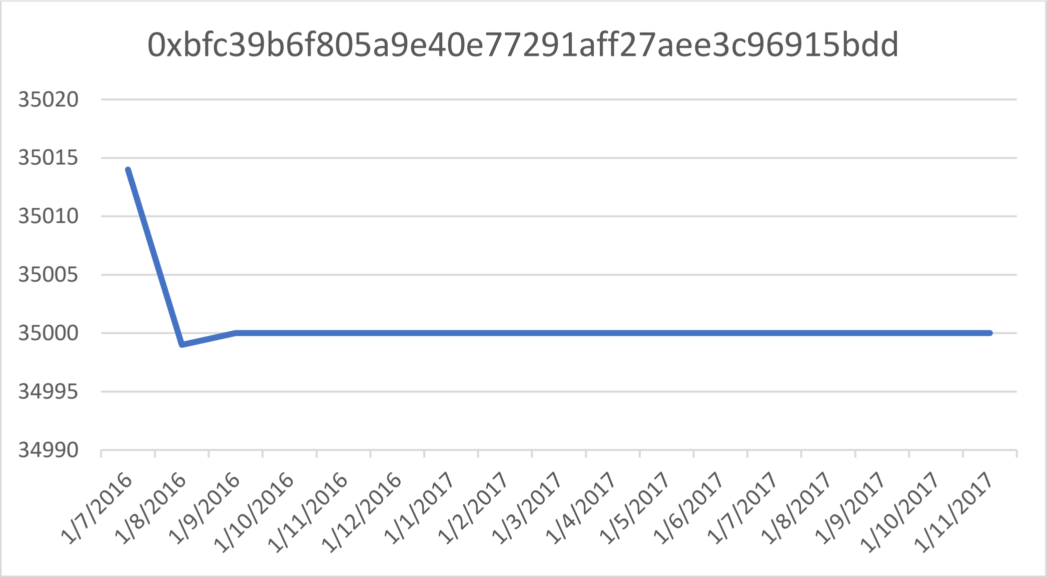


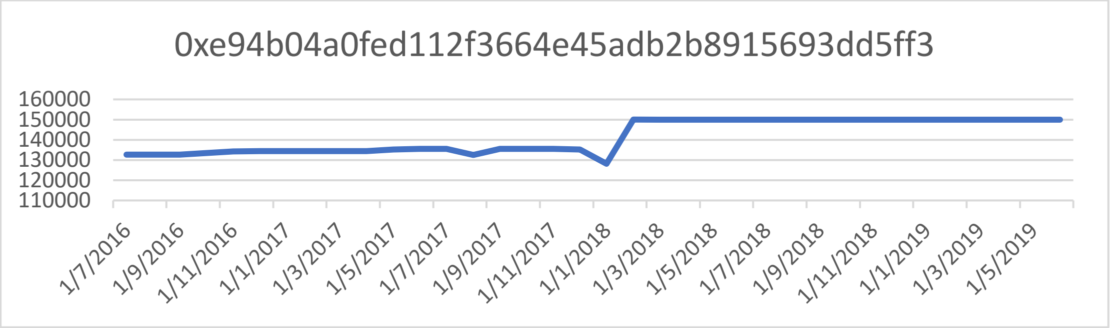
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
Date
1/8/2016
1/10/2016
1/12/2016
1/2/2017
1/4/2017
1/6/2017
1/8/2017
1/10/2017
1/12/2017
1/2/2018
1/4/2018
1/6/2018
1/8/2018
1/10/2018
1/12/2018
1/2/2019
1/4/2019
1/6/2019

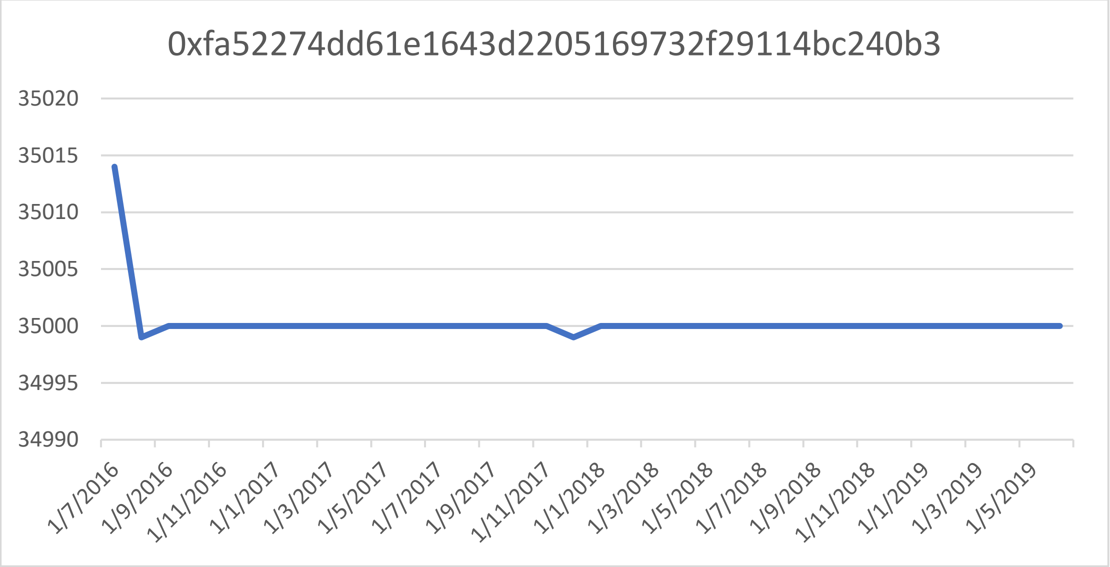

0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
1/7/2016
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019


0xabbb6bebfa05aa13e908eaa492bd7a8343760477
160000
140000
120000
100000
80000
60000
40000
20000
0
1/9/2016
1/11/2016
1/1/2017
1/3/2017
1/5/2017
1/7/2017
1/9/2017
1/11/2017
1/1/2018
1/3/2018
1/5/2018
1/7/2018
1/9/2018
1/11/2018
1/1/2019
1/3/2019
1/5/2019


0xbb9bc244d798123fde783fcc1c72d3bb8c189413
35020
35015
35010
35005
35000
34995
34990
1/7/2016
1/8/2016
1/9/2016
1/10/2016
1/11/2016
1/12/2016
1/1/2017
1/2/2017
1/3/2017
1/4/2017
1/5/2017
1/6/2017
1/7/2017
1/8/2017
1/9/2017
1/10/2017
1/11/2017



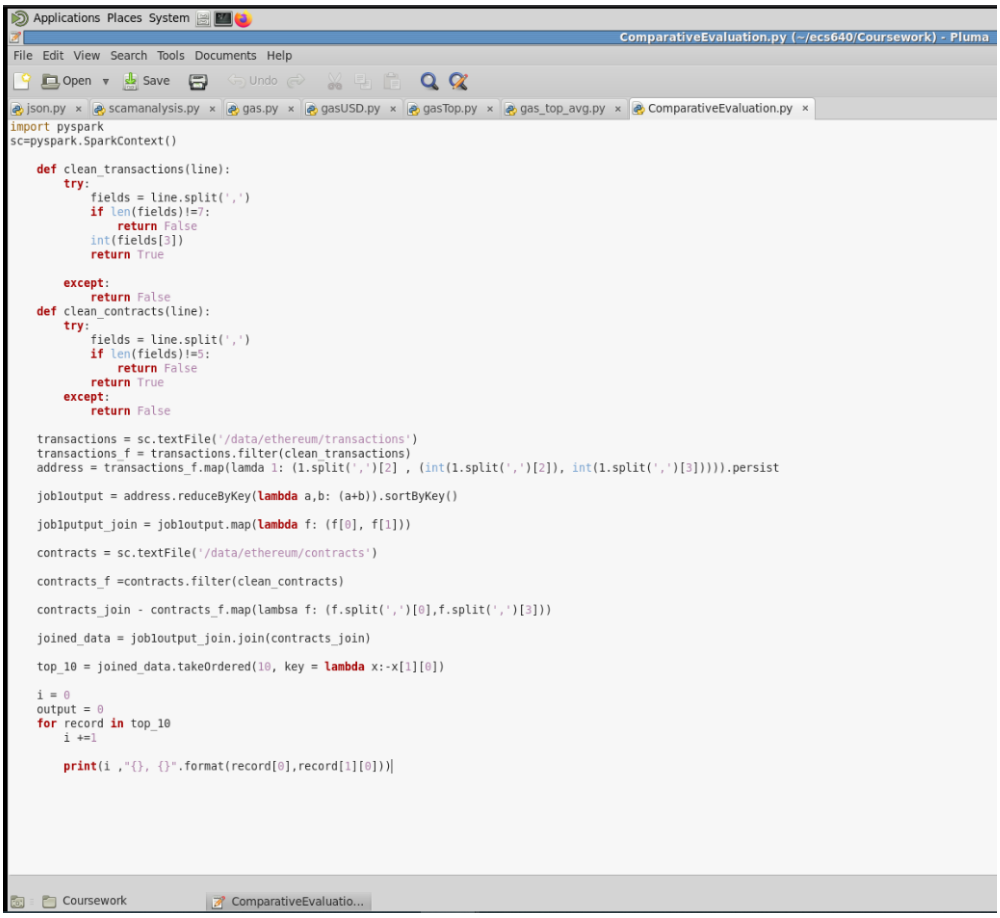




- In each table, the complexity of gas requirements is seen by the top 10 smart contracts. Few smart contracts consume more gas over time, but few contracts require less gas over time.3

Comparative Evaluation

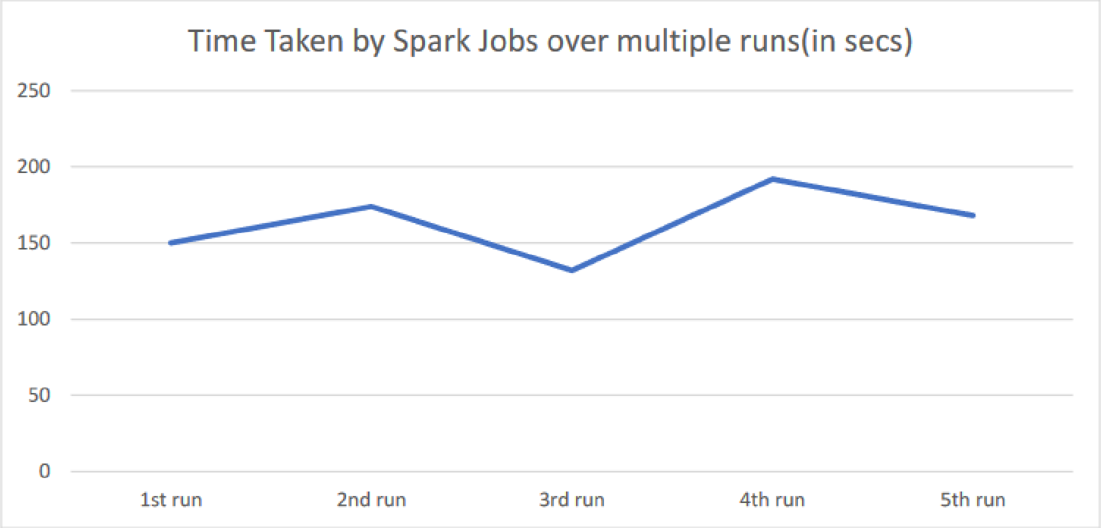
Code: Program name is comperativeEvaluation.py

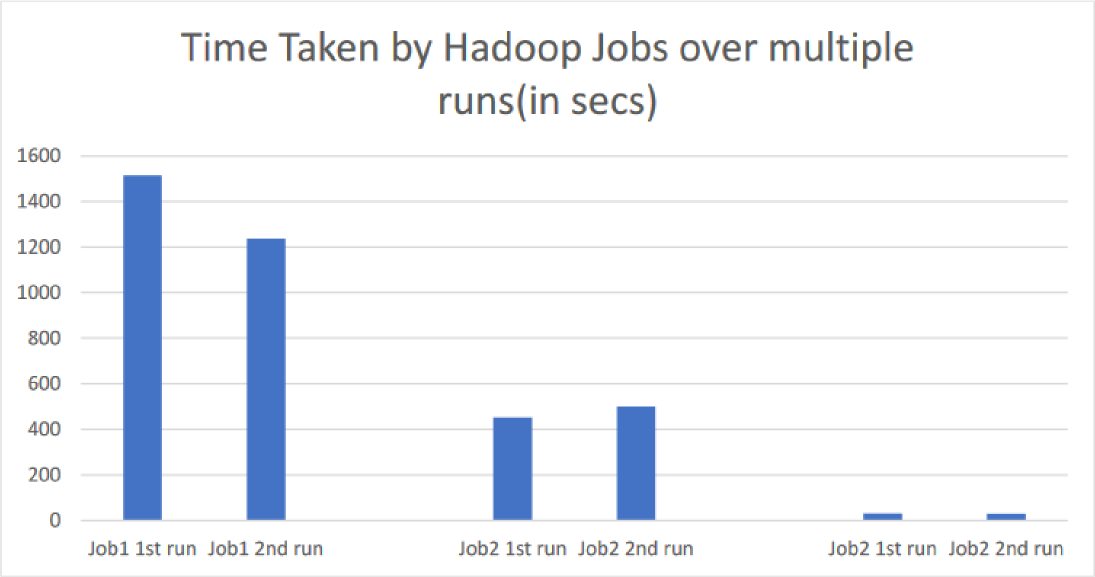


```
import pyspark
sc=pyspark.SparkContext()

    def clean_transactions(line):
        try:
            fields = line.split(',')
            if len(fields)!=7:
                return False
            int(fields[3])
            return True

        except:
            return False
    def clean_contracts(line):
        try:
            fields = line.split(',')
            if len(fields)!=5:
                return False
            return True
        except:
            return False

    transactions = sc.textFile('/data/ethereum/transactions')
    transactions_f = transactions.filter(clean_transactions)
    address = transactions_f.map(lamda 1: (1.split(',')[2] , (int(1.split(',')[2]), int(1.split(',')[3])))).persist

    job1output = address.reduceByKey(lambda a,b: (a+b)).sortByKey()

    job1putput_join = job1output.map(lambda f: (f[0], f[1]))

    contracts = sc.textFile('/data/ethereum/contracts')

    contracts_f =contracts.filter(clean_contracts)

    contracts_join - contracts_f.map(lambsa f: (f.split(',')[0],f.split(',')[3]))

    joined_data = job1output_join.join(contracts_join)

    top_10 = joined_data.takeOrdered(10, key = lambda x:-x[1][0])

    i = 0
    output = 0
    for record in top_10
        i +=1

        print(i ,"{}, {}".format(record[0],record[1][0]))
```

Code Explanation:

The first line reads the dataset of transactions using the sparkcontext feature of pyspark. Using the user-defined function clean transactions, the second line filters any bad lines from the transaction dataset. We map the key as address and value as values from the transaction dataset in the third line using the lambda function of spark. We use the spark reductionByKey feature in the fourth row to measure the transaction value aggregate. This output is stored in memory. We map the key as an address and value as combined values from the previous operation output in the fifth row. In-memory processing by Spark is made use of here. These values are stored in the job1output join variable.•The contract dataset reads from the sixth line. The seventh line filters the bad lines from the dataset of contracts. The eighth line maps the key as an address and value as a block number from the dataset of contracts using the lambda function of spark. Using the spark join operation, the ninth line conducts the join operation. The output of the joined dataset is the vector joined data.•Tenth line performs the sorting process and uses the takeOrdered function of spark to filter only 10 values. In the lambda function x:-x[1][0], the symbol'-' sorts the values in decreasing order, then leaving the highest value. The final move is to format the values after sorting and printing the top 10 services.





Spark jobs seem to perform faster for this task, comparing the outcomes of Hadoop and Spark jobs. This is due to Spark's in-memory processing where the intermediate outcomes can be stored in RDD and the next transformation or action can be applied to that RDD. The output data must be written and read to HDFS every time we run Hadoop jobs, but we can read it directly from RDD in Spark. In a single Spark Job, all three jobs of Part B will be implemented. With this task, several maps and reduction steps can be implemented in a single Hadoop job, but there would be more time required to manage the documents through map reduction work.

The amount of shuffle and sort that occurs during the join procedure is high, reducing the efficiency of the job. Looking at the graphs the average time taken by Spark job is 163 seconds and Hadoop jobs is 627 seconds. So it is clear that Spark jobs perform better than Hadoop mar reduce jobs.