Coursework : Ethereum Analysis

Name: Muhammad Hassaan Anwar

Date Submitted: 2nd December - **9 :55 AM** (+5 marks worthy)

Student Number: 190754334

PART A. TIME ANALYSIS

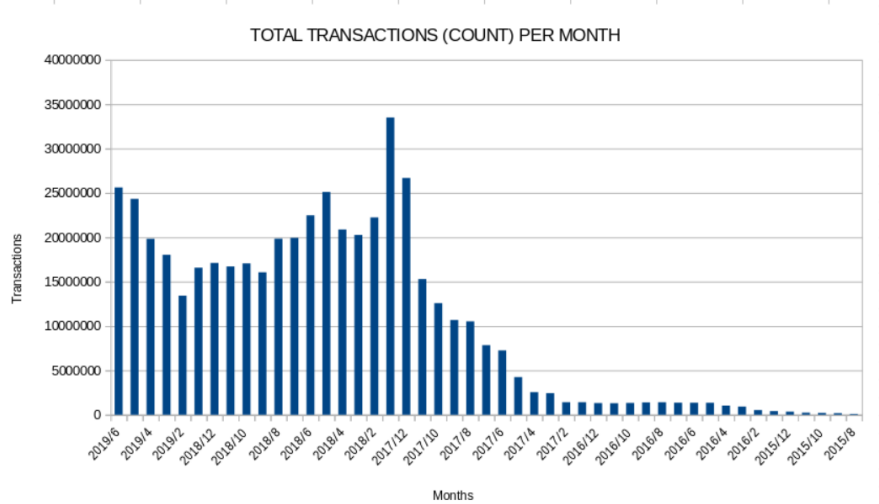
Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

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.)

GRAPH:

2019/6	25613628
2019/5	24332475
2019/4	19830158
2019/3	18029582
2019/2	13413899
2019/1	16569597
2018/12	17107601
2018/11	16713911
2018/10	17056926
2018/9	16056742
2018/8	19842059
2018/7	19937033
2018/6	22471788
2018/5	25105717
2018/4	20876642
2018/3	20261862
2018/2	22231978
2018/1	33504270
2017/12	26687692
2017/11	15292269
2017/10	12570063
2017/9	10672734
2017/8	10523178
2017/7	7835875
2017/6	7244657
2017/5	4245516
2017/4	2539966
2017/3	2426471
2017/2	1410048
2017/1	1409664
2016/12	1316131
2016/11	1301586
2016/10	1329847
2016/9	1387412
2016/8	1405743
2016/7	1356907
2016/6	1351536
2016/5	1346796
2016/4	1023096



ANALYSIS:

As we can see from the bar plot above, we can see that the number of transactions were lesser in the years 2015, 2016. In February 2017 we see a gradual increase in transactions, which reached there maximum at January 2018. Following after that, we see a gradual decline in February, March , April , but in May 2018 we see an increase in the transactions.

Following after that, we see a decrease in number of transactions until February 2019. Onwards, Ethereum transactions begin to stabilize again as the trend goes upward back again.

JOB ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1574975221160_4880

Time: 1 min 40 sec

CODE:

MAPPER:

```
def mapper(self, _, line):
    try:

        fields = line.split(",")
        if (len(fields)==7):
        #access the fields you want, assuming the format is correct now

            time_epoch = int(fields[6])
            date=time.gmtime(time_epoch)
            month=date.tm_mon
            year=date.tm_year
            x=(month,year)
            yield(x,1)

    except:
        pass
```

In the mapper field, time (month and year) will be used as the key, and for each transaction made, 1 will be used as the value.

In the reducer, for each "month,year" pair, all of its values would be summed.

A combiner is added to speed up our execution process.

```
def combiner(self,day,counts):

    # print("total words appearing more than 10 times = "+count(word))
    yield(day,sum(counts))

def reducer(self,day,counts):
    #you have to implement the body of this method. Python's sum() function will
probably be useful

    # print("total words appearing more than 10 times = "+count(word))
    yield(day,sum(counts))
```

With each month, the values of transactions that happened in that month are summed in the reducer.

A combiner is also added to speed up the performance of map reduce operations.

PART B. TOP TEN MOST POPULAR SERVICES

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. This is, however, only one possibility, with several other viable ways of completing this assignment.

JOB 1 - INITIAL AGGREGATION

To workout which services are the most popular, you will first have to aggregate transactions to see how much each address within the user space has been involved in. You will want to aggregate value for addresses in the to_address field. This will be similar to the wordcount that we saw in Lab 1 and Lab 2.

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_address field 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 within contracts this should be filtered out as it is a user address and not a smart contract.

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, utilising what you have learned from lab 4.

JOB 1 - ID:

http://andromeda.student.eecs.qmul.ac.uk:19888/jobhistory/job/job_1574975221160_2908

Time Taken: 00hrs, 26mins, 18sec

JOB 2 - ID:

job_1574171293853_4851 00hrs, 06mins, 44sec

Job3 - ID:

job_1574171293853_4864 00hrs, 00mins, 23sec

CODE JOB 1:

In the mapper, we need the **to_address** from the transactions table as the key, and the **values** from the values column from the transactions table as the values

```
def mapper(self, _, line):
    try:
```

```
        fields = line.split(",")
        if (len(fields)==7):
        #access the fields you want, assuming the format is correct now

            x=fields[2]
            y=int(fields[3]) #value

            yield(x,y)

    except:
        pass
```

```
def combiner(self,address,counts):

    yield(address,sum(counts))

def reducer(self,address,counts):

    yield(address,sum(counts))
```

In the reducer, for each address, all of its transactions (independent of time-stamp) that appeared in the transactions table would be summed.

A combiner is also added to speed up our job execution speed.

For our second job, A repartition join is required (since both files are too large to be considered for a replication join) between the aggregate of the first job and the contracts table. Since non smart-contracts addresses are needed to be filtered out:

CODE For Job2:

I decided to do job 2 and job 3 in the same python file, using MRStep.

```
def steps(self):
    return [MRStep(mapper=self.mapper_join_init,
                   reducer=self.reducer_transactions),
            MRStep(mapper=self.mapper_new,
                   reducer=self.reducer_new)]
```

MAPPER:

This is a repartition join, this code will run on the aggregate of job1 and contracts table.

My aggregate transactions file was ** **tab seperated txt file**** having only 2 fields.

The contracts table has 5 fields and is a **CSV** format.

The code will run on both files simultaneously, taking each file at a time. Comparing the lengths, if length is 2 then it means that it is running on the aggregate of transactions, if length is 5, it means that it is contracts table.

The aggregates file (Output of Job1) look like this:

```
"0x0000000000000000000000000000000000000003"    169348682492768122
"0x0000000000000000000000000000000000000005"    172226791168647579
"0x0000000000000000000000000000000000000007"    1
"0x0000000000000000000000000000000000000009"    1
"0x000000000000000000000000000000000000000a"    1000000000000001
"0x000000000000000000000000000000000000000c"    51227906993421169
"0x000000000000000000000000000000000000000e"    1
```

When running on the aggregates file, it will take the addresses as the key, and aggregate values as the values. The value would be passed with a '1' to make it distinguishable in the reducer.

The Contracts file look like this:

```
0xee5b22b526dc9ba9d0d3a998cd2755eb4fbfd511,false,false,8623675,2019-09-26
09:22:13 UTC
0x7de34c3203a1e45e8c57427f5bc8a44446d110b9,false,false,8627300,2019-09-26
22:51:07 UTC
0x5fa80c8bffcf5e5ba1a0a690ed017d5be023c843,false,false,8621217,2019-09-26
00:07:26 UTC
0x065b40866b01c30e766810b55430b8983821f8d3,false,false,8621873,2019-09-26
02:34:15 UTC
0x24cd2cb9c32d21b3115df0f0d75ede886d263d1b,false,false,8621194,2019-09-26
00:00:57 UTC
0x1ff2f1efc26c17ef1bf0ceaae6e9f8e1888a5b5d,false,false,8621366,2019-09-26
00:37:48 UTC
0x3b79047d6b19b461d07a2faa416fdcc8dd2f18ed,false,false,5359623,2018-04-01
07:18:29 UTC
0x4bd910bfb961c51bb1c1eeaafbcf2b53fbd774f2,false,false,5359892,2018-04-01
08:20:12 UTC
0x8cf8634d92bbab940563e5908a1bb11f33a0031a,false,false,5359637,2018-04-01
07:21:10 UTC
0x0000000000000000000000000000000000000014,false,false,5359391,2018-04-01
06:20:13 UTC
```

When running on the contracts table it will take addresses as the value, and any other column from the table as the value (taking a value in the contracts table is not necessary, it is just taken for the sake of code format). The value would be passed with a '2' to make it distinguishable in the reducer.

```
def mapper_join_init(self, _, line):
    try:

        #one mapper, we need to first differentiate among both types
        if(len(line.split('\t')))==2:

            fields = line.split('\t')
            #this should be a company sector line
            join_key = fields[0].replace('"','')
            join_value = int(fields[1])
            yield (join_key, (join_value,1))

        elif(len(line.split(','))==5:
```

```
            fields = line.split(',')
            #this should be a base closing price line
            join_key = fields[0]
            join_value = int(fields[3])
            yield (join_key,(join_value,2))

    except:
        pass
```

REDUCER:

In the Reducer section two variable are initialized, value_reducer which is an **integer** and **check**.

If an address has its presence in both of the tables, it's value would be stored in a variable, and the check would be assigned the boolean **True**.

Now, for each address having a boolean greater than 2, and having an aggregate value that is greater than 0, we will yield it's address and value.

```
def reducer_transactions(self, to_address, values):

    value_reducer=0
    check=False

    for value in values:
        if value[1] == 1:
            value_reducer=value[0]
        elif value[1]==2:
            check=True

    if check==True and value_reducer>0:
        x=(to_address,value_reducer)
        yield(None,x)
```

CODE for Job3:

The Values are sorted in the reducer, and top10 values are yielded.

```
def mapper_new(self, _,x):
    yield(None,x)

def reducer_new(self, _,vals):
    sorted_values = sorted(vals, reverse=True, key = lambda x: x[1])
    for x in range(10):
        yield(x+1,'{}-{}'.format(sorted_values[x][0],sorted_values[x][1]))
```

TOP 10:

```
1 0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444-84155100809965865822726776
2 0xfa52274dd61e1643d2205169732f29114bc240b3-45787484483189352986478805
3 0x7727e5113d1d161373623e5f49fd568b4f543a9e-45620624001350712557268573
4 0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef-43170356092262468919298969
5 0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8-27068921582019542499882877
6 0xbfc39b6f805a9e40e77291aff27aee3c96915bdd-21104195138093660050000000
7 0xe94b04a0fed112f3664e45adb2b8915693dd5ff3-15562398956802112254719409
8 0xbb9bc244d798123fde783fcc1c72d3bb8c189413-11983608729202893846818681
9 0xabbb6bebfa05aa13e908eaa492bd7a8343760477-11706457177940895521770404
10 0x341e790174e3a4d35b65fdc067b6b5634a61caea-83790007519177556240575000
```

PART C:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574171293853_6780/

Streamjob7011564441740629895.jar

Gas Guzzler:

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? How does this correlate with your results seen within Part B

JOB ID:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_5608/

CODE:

```
def mapper(self, _, line):
    try:

        fields = line.split(",")
        if (len(fields)==7):
        #access the fields you want, assuming the format is correct now

            time_epoch = int(fields[6])
            gas_price=int(fields[5])
            date=time.gmtime(time_epoch)
            month=date.tm_mon
            year=date.tm_year
            x=(month,year)
            yield(x,(gas_price,1))

    except:
        pass
```

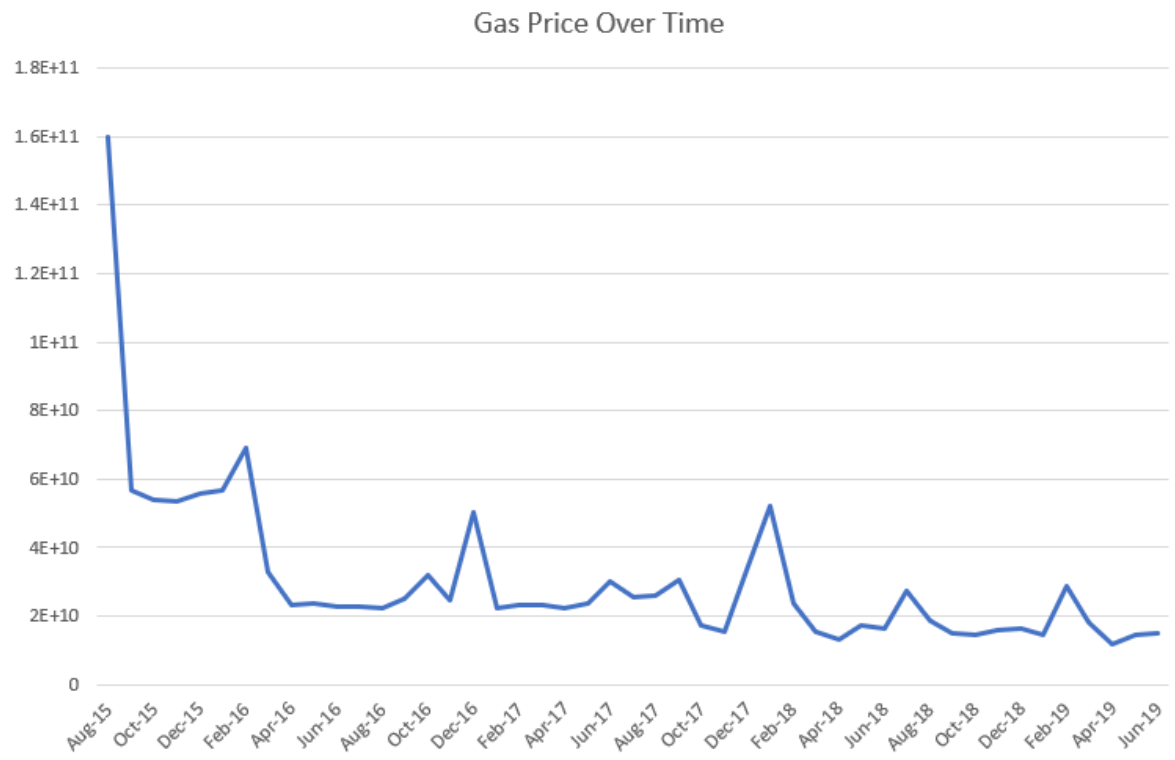
In the mapper, each month and year pair is taken as the key, and it's gas_prices for all of the transactions from the transactions tables are yielded as the value.

```
def reducer(self,month,gas):
    #you have to implement the body of this method. Python's sum() function will
probably be useful
    count1=0
    count2=0
    for x in gas:
        count1+=int(x[0])
        count2+=int(x[1])
    average=count1/count2

    # print("total words appearing more than 10 times = "+count(word))
    yield(month,average)
```

In the reducer, for each month, it's gas prices are averaged.

GAS PRICE OVER TIME:



In August 2015. The value of the gas price was at the highest point. With further months, the gas prices followed a downward trend .

Between February and April 2016 the gas prices started increasing again, followed by downward trends in June, July and August 2016.

The gas prices started increasing back in December 2016, this surge lasted for 2 months. Followed by stability in prices. In December 2017 the gas prices rose up again, this surge also lasted for just 2 months.

The prices are stable until June 2019.

JOB ID's used to workout complexity:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_0214/

http://andromeda.student.eecs.qmul.ac.uk:19888/jobhistory/job/job_1574975221160_1624

http://andromeda.student.eecs.qmul.ac.uk:19888/jobhistory/job/job_1574975221160_1630

CODE

To workout the complexity, I have made a replication join between top 10 contracts and transactions from Ethereum dataset.

REPLICATION JOIN BETWEEN TOP10 Contracts and Transactions table:

```
sector_table = {}

def steps(self):
    return [MRStep(mapper_init=self.mapper_join_init,
                   mapper=self.mapper_repl_join),
            MRStep(mapper=self.mapper_length,
                   reducer=self.reducer_sum)]
```

The top 10 contracts are used to initialize the Replication join.

```
def mapper_join_init(self):
    # load companylist into a dictionary
    # run the job with --file input/companylist.tsv

    with open("partb.txt") as f:
        for line in f:
            fields = line.split("-")
            key = fields[0]
            val = fields[1]
            self.sector_table[key] = val


 # Using the transactions table as the other table to be joined with.

# def mapper_repl_join(self, _, line):
def mapper_repl_join(self, _, line):
    fields = line.split(",")
    address=fields[2]
    date=fields[6]
    gas=fields[4]
    value=fields[3]
```

If a match exists, we will yield the time stamp, gas and value for the transaction made by the contract.

```
    for x in self.sector_table:

        if address == x:
            tiime=time.strftime("%Y-%m-%d",time.gmtime(int(date)))
            pair=(tiime,gas,value)
            yield(address,pair)
            break
```

```
def mapper_length(self,key,values):
    yield(key,list(values))

def reducer_sum(self, key,values):
    for x in values:
        yield(key,x)
```

This will give us a list of addresses and date of transaction as the key, and gas and values as the values.

We still need to average them.

AVERAGING VALUES AND GAS WITH TIME:

```
def mapper(self, _, line):
```

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

        fields=line.split('\t')
        address=fields[0].replace('"','')
        x=fields[1].replace('[','')
        z=x.replace(']','')
        y=z.split(',')
        gas=y[1].replace('"','')
        value=y[2].replace('"','')
        time=y[0][1:8]
        yield ((address,time),(int(gas),int(value)))
        # yield((address,time),int(value))

def reducer(self, keys, values):

    count = 0
    gastotal = 0
    valuetotal=0
    for value in values:
        count += 1
        gastotal += int(value[0])
        valuetotal+=int(value[1])
    gasAverage=gastotal/count
    valueAverage=valuetotal/count
    yield (keys,(gasAverage,valueAverage))
```

Output Format:

Address , Transactions grouped by month, Average Gas per month, Value average in that month

```
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", "2018-08"]   [49989.23029457137,
5.028670614390656e+18]
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", "2018-11"]   [49931.85593220339,
8.612298791166551e+18]
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", "2019-01"]   [49998.24177300202,
8.116597447770316e+18]
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", "2019-03"]   [50018.317254854075,
1.7068770101951398e+19]
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", "2019-05"]   [50015.62154466722,
2.5472513240051016e+19]
["0x341e790174e3a4d35b65fdc067b6b5634a61caea", "2016-08"]   [88888.0,
5.75614e+20]
["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", "2016-08"]   [47008.0,
2.1735242876835676e+20]
["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", "2016-11"]   [47008.0,
1.5959357464364112e+20]
["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", "2017-01"]   [47008.0,
1.960146105096436e+20]
["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", "2017-03"]   [47008.0,
4.910706966667134e+20]
["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", "2017-05"]   [47008.0,
2.4936340064773476e+20]
```

Working out the complexity :

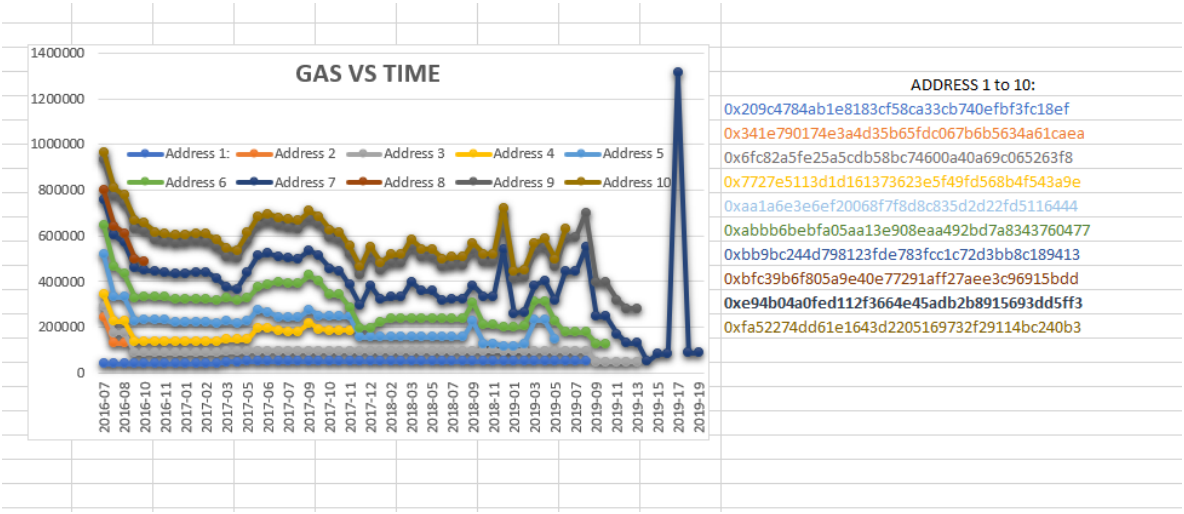
To work out the complexity of our transactions and to see if they have became more complex with time or not.

I made a replication join of my top 10 values contracts from Part-B with the transactions table, to find their history of transactions.

For each top-10 address, its time-stamp and gas has been yielded.

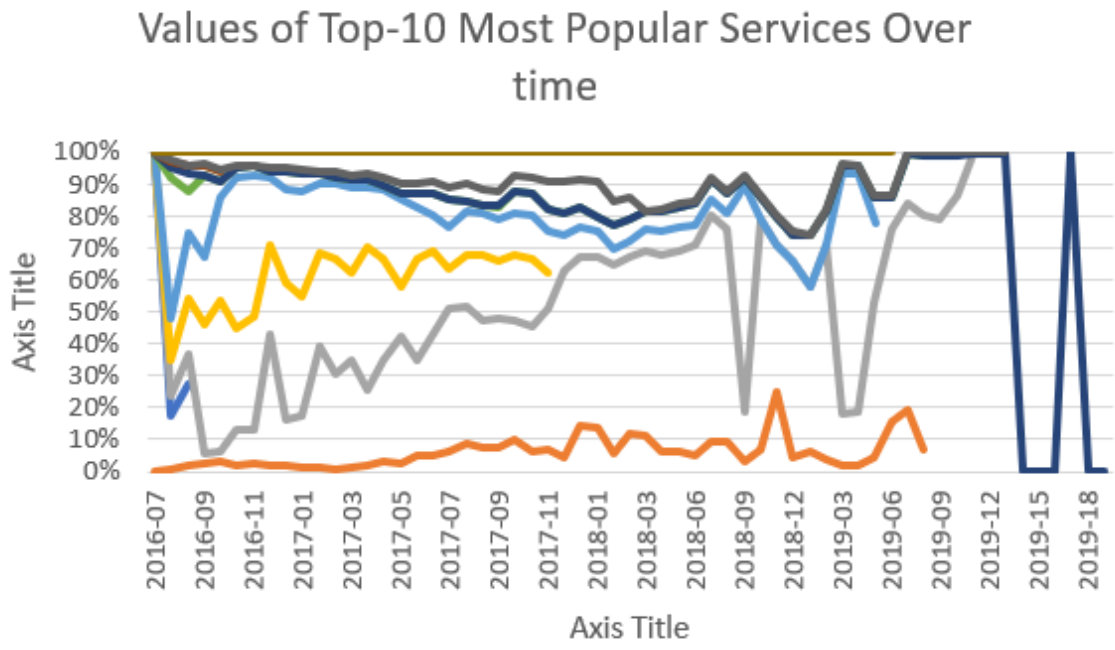
PLOT:

GAS VS TIME:



All of the contracts follow a similar trend, the graph shows that gas used by the contracts is increasing along the time scale.

To see if there is a relation between gas prices and the output of part B, and if they correlate or not. I have also plotted the aggregate of Transaction values with time. To check if Higher transactions lead to higher gas and complexity of the algorithm.



And the answer to the question is **YES**

When I matched the trend of values and its gas consumption for each contract in both graphs. I found that they follow the same pattern along the period of time.

It seems that, since the contract's transactions values are increasing with time, the complexity is increasing with time too.

Popular Scams: Utilising the provided scam dataset, what is the most lucrative form of scam? How does this change throughout time, and does this correlate with certain known scams going offline/inactive? (20/30)

JOB IDs Used To work out Lucrativeness Of Scams:

STEP1 :
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_5781/
STEP2:
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1574975221160_5806

IDEA BEHIND WRITING THE CODE:

To work out most lucrative form of scams. A replication join would be made between Aggregate of transactions from Part_B job 1 output to the Scams.json file. To see if those addresses were involved in any sort of scams. The next step would be aggregating the scammed contracts with respect to their types and values of scams.

Code:

Using 2 steps

```
def steps(self):
    return [MRStep(mapper_init=self.mapper_join_init,
                   mapper=self.mapper_repl_join),
            MRStep(reducer=self.reducer_sum)]
```

Initializing .json file to be mapped.

A dictionary with name sector_table is also initialized.

They Keys of the dictionary would be the addresses from the .json file and the values would be the categories from the .json file.

```
sector_table={}
def mapper_join_init(self):
    with open('scams.json', 'r') as f:
        file = json.load(f)
    for x in file['result']:
            key = x
            val = file['result'][x]['category']
            self.sector_table[key] = val
```

Now a replication join would be made with the addresses.

If an address is found in both the tables. Its category of Scam and its value aggregate of transaction would be yielded.

The Category of Scam would be they Key and the value of transaction would be the value.

```
def mapper_repl_join(self, _, line):
    fields = line.split("\t")
    address=fields[0].replace('"','')
    value=int(fields[1])
    for x in self.sector_table:
        if address == x:
            category=self.sector_table[x]
            yield(category,value)
            break
```

For each category of scam, its aggregate values would be summed.

```
def reducer_sum(self, key,values):
    yield(key,sum(values))
```

OUTPUT:

"Scamming" 38407781260421703730344
"Fake ICO" 135645756688962997967 8
"Phishing" 26927757396110618476458

The above output clearly means that " Scamming " was the most lucrative scam.

Scam Trends with Time:

Job1: http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_5839/

Job2: http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_5849/

IDEA BEHIND WRITING THE CODE:

For the first job, the scams.json file would go through a replication join with the complete transactions data the dataset. If the addresses involved in the scams.json file are present in the transactions table. Their values, timestamp and category would be yielded.

In the next step, for each category of scam , all of the transactions that were carried out in a particular month would be summed.

CODE :

The same intuition is followed in this code too.

```
sector_table = {}
def steps(self):
    return [MRStep(mapper_init=self.mapper_join_init,
                   mapper=self.mapper_repl_join),
            MRStep(mapper=self.mapper_length,
                   reducer=self.reducer_sum)]
```

```
 def mapper_join_init(self):
```

```
    with open('scams.json', 'r') as f:
        file = json.load(f)
    for x in file['result']:

            key = x
            val = file['result'][x]['category']
            self.sector_table[key] = val
```

```
def mapper_repl_join(self, _, line):
    try:
        fields = line.split(",")
        if len(fields) == 7:
            address=fields[2]
            value=int(fields[3])
            time_epoch = fields[6]
            transac = time.strftime("%Y-%m",time.gmtime(int(time_epoch)))
            for x in self.sector_table:
                if address == x:
                    category=self.sector_table[x]
                    yield((transac,category),value)
                    break
                else:
                    pass
    except:
        pass
```

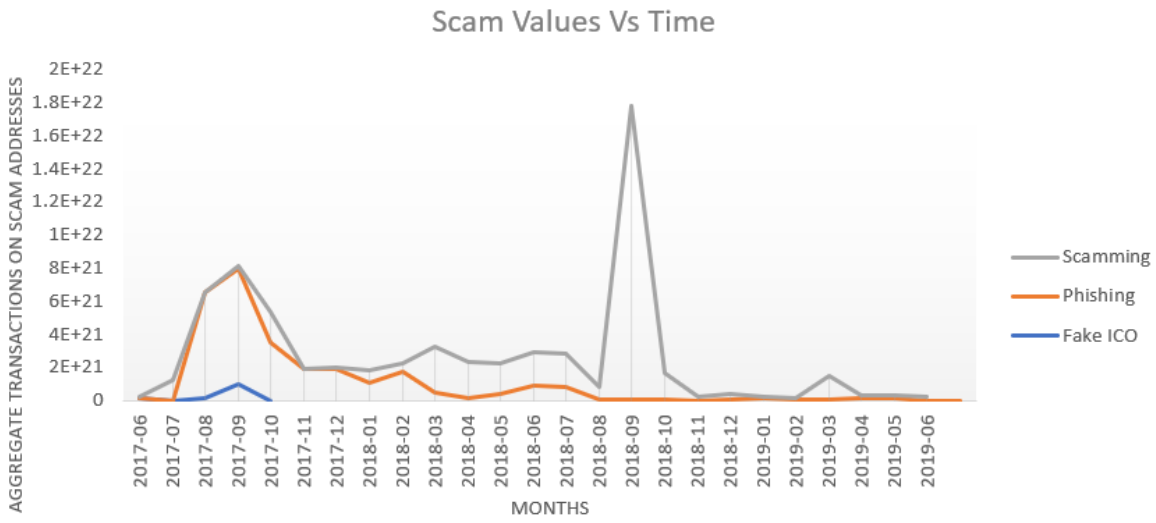
```
def mapper_length(self,key,values):
    yield(key,values)

def reducer_sum(self, key,values):
    yield(key,sum(values))
```

OUTPUT:

["2017-06", "Fake ICO"] 18267402332376326800
["2017-06", "Phishing"] 1000000000000000000
["2017-07", "Scamming"] 1238944359754270060501
["2017-08", "Fake ICO"] 181164662377136166221
["2017-08", "Phishing"] 6974984846564749956068
["2017-09", "Scamming"] 181698633896218700114
["2017-10", "Scamming"] 1839392288663253070196
["2017-11", "Phishing"] 1931130818121393033801
["2017-12", "Scamming"] 27509581908918747084
["2018-01", "Phishing"] 1768033097561016633043
["2018-02", "Scamming"] 498402535182915198977
["2018-03", "Phishing"] 110405318464457062612
["2018-04", "Scamming"] 2234444404319805847088
["2018-05", "Phishing"] 888692428537232943877
["2018-06", "Scamming"] 1994605998015701010629
["2018-07", "Phishing"] 91317597587664240011
["2018-08", "Scamming"] 732794263202598956076
["2018-09", "Phishing"] 27232229905500000000
["2018-10", "Phishing"] 13457963543301563693

GRAPH:



As it is inferred from the plot above,

Fake ICO scam had a really short life span. Getting Initialized in July 2017 and Ending in October 2017.

For the Phishing, it was initialized in June 2017, surged up high in September and October of 2017.

Scamming initialized in June 2016, surging up high and profiting the most since January 2011. Scamming had its all time high value in September 2018.

As seen from the analysis, Scamming remains to be the most lucrative one, over a long period of time.
