BIG DATA PROCESSING

COURSEWORK – ETHEREUM ANALYSIS

ANALYSIS OF ETHEREUM TRANSACTIONS AND SMART CONTRACTS

ASSIGNMENT

Write a set of Map/Reduce (or Spark) jobs that process the given input and generate the data required to answer the following questions:

PART A. TIME ANALYSIS (30%)

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.)

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

def clean_transactions(line):
    try:
        fields = line.split(',')
        if len(fields)!=7:
            return False
        float(fields[6])
        return True
    except:
        return False

def get_monthYear(line):

    fields=line.split(',')
    timestamp = int(fields[6])
    monthYear = time.strftime("%m-%Y",time.gmtime(timestamp))
    return(monthYear,1)

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

clean_lines = transactions.filter(clean_transactions)
keyMY=clean_lines.map(get_monthYear).persist()
output=keyMY.reduceByKey(lambda a,b: a+b).sortByKey()

inmem=output.persist()
inmem.saveAsTextFile("/user/rkt31/partAoutput")
```

Define a function clean_transactions to clean the transactions dataset.

Split the lines into fields and check if there are all 7 fields else return False. This makes sure that lines with missing columns will be omitted.

Define a function to get the month and year. Field 6 of the transactions dataset contains the timestamp. Using built-in function **time** we are extracting th month and year from the timestamp and returning it.

Extracting transactions from ethereum dataset into a variable called transactions

Applying filter on the transactions dataset using the defined function clean_transactions to get clean lines that is the ones without missing fields

Map month and year of the filtered transactions with monthYear as key and value 1 and store it in variable keyMY. Then reduce the key-value pairs to get the count of number of transactions occurred in a particular month and year.

JOB ID :
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_2979

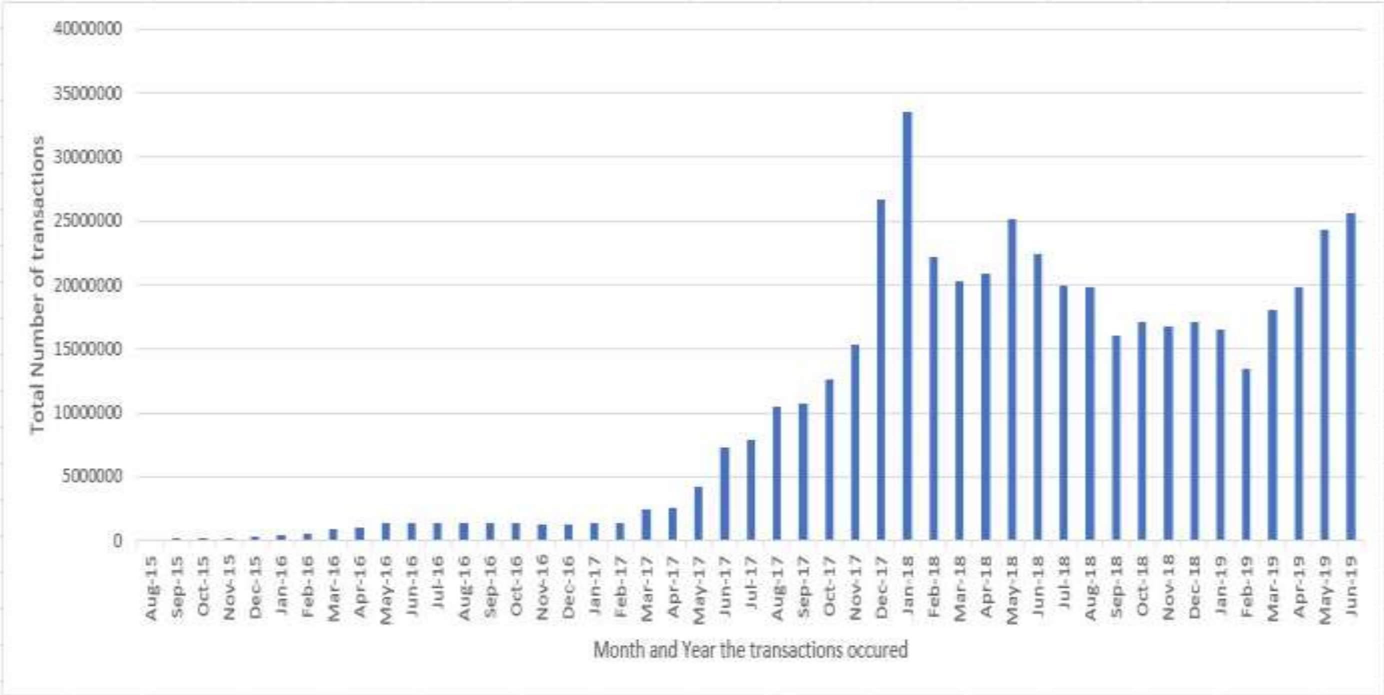
The raw output for the above code is as follows which yields the number of transactions occurred in a particular month of the year from the start and end of the Ethereum dataset.

partAoutput.txt - Notepad

File Edit Format View Help

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

A Bar plot of the number of transactions occurred every year from the start and end of the dataset was plotted in Excel as below:



PART B. TOP TEN MOST POPULAR SERVICES (40%)

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.

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

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

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

transactions = sc.textFile("/data/ethereum/transactions")
trans_filtered = transactions.filter(clean_transactions)
address=trans_filtered.map(lambda l: (l.split(',')[2], int(l.split(',')[3]))).persist()
job1output = address.reduceByKey(lambda a,b:(a+b))
job1output_join=job1output.map(lambda f:(f[0], f[1]))

contracts = sc.textFile("/data/ethereum/contracts")
contracts_filtered = contracts.filter(clean_contracts)
contracts_join = contracts_filtered.map(lambda f: (f.split(',')[0],f.split(',')[3]))

job2output =job1output_join.join(contracts_join)
top10=job2output.takeOrdered(10, key = lambda x:-x[1][0])
for record in top10:
    print("{}: {}".format(record[0],record[1][0]))
```

Defining functions clean_transactions and clean_contracts to clean transactions and contracts dataset by yielding only the datasets having all fields that is 7 and 5 respectively

The filtered transactions data set is split into fields and mapped using to_address as the key and value field as value. This is done using the lambda function l.

Then the aggregation of to_address and value is obtained by the use of reducer.

The filtered contracts will be mapped using address i.e field [0] as key and block number I.e field[3] as value

The to_address of transactions are joined with address of contracts

The top 10 smart contracts are obtained using the built-in function takeOrdered. It is ordered in a decending order using -x[]

JOB ID :
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_3492

The top 10 smart contracts are shown below in descending order

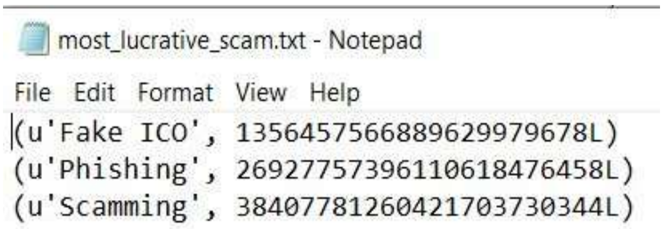
```
rkt31@itl100 ~/ec19488/coursework> spark-submit partBSpark.py
19/12/04 22:40:49 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
19/12/04 22:40:49 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disabled.
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444: 84155100809965865822726776
0xfa52274dd61e1643d2205169732f29114bc240b3: 45787484483189352986478805
0x7727e5113d1d161373623e5f49fd568b4f543a9e: 45620624001350712557268573
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef: 43170356092262468919298969
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8: 27068921582019542499882877
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd: 21104195138093660050000000
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3: 15562398956802112254719409
0xbb9bc244d798123fde783fcc1c72d3bb8c189413: 11983608729202893846818681
0xabbb6bebfa05aa13e908eaa492bd7a8343760477: 11706457177940895521770404
0x341e790174e3a4d35b65fdc067b6b5634a61caea: 8379000751917755624057500
rkt31@itl100 ~/ec19488/coursework>
```

PART C. DATA EXPLORATION (30%)

SCAM ANALYSIS

1. **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 ID :
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_4066
OUTPUT



most_lucrative_scam.txt - Notepad

File Edit Format View Help

```
(u'Fake ICO', 1356457566889629979678L)
(u'Phishing', 26927757396110618476458L)
(u'Scamming', 38407781260421703730344L)
```

From the output file most_lucrative_scam.txt we can see that the most lucrative scam is Scamming as it has a greater number of transactions.

The change in most lucrative scam over time can be analysed using the output file timeanalysis.txt

timeanalysis.txt - Notepad

File Edit Format View Help

```
((u'Fake ICO', '06-2017'), 182674023323763268000L)
((u'Fake ICO', '06-2018'), 1238318290000000000)
((u'Fake ICO', '07-2017'), 16242199484949186112L)
((u'Fake ICO', '08-2017'), 181164662377136166221L)
((u'Fake ICO', '09-2017'), 975138363413781359345L)
((u'Phishing', '01-2018'), 1768033097561016633043L)
((u'Phishing', '01-2019'), 45532540234953645569L)
((u'Phishing', '02-2018'), 467876789232524118169L)
((u'Phishing', '02-2019'), 75960801308673932726L)
((u'Phishing', '03-2018'), 110405318464457062612L)
((u'Phishing', '03-2019'), 146871180350125628000L)
((u'Phishing', '04-2018'), 420285984122934750845L)
((u'Phishing', '04-2019'), 161651973926857053455L)
((u'Phishing', '05-2017'), 90000000000000000)
((u'Phishing', '05-2018'), 888692428537232943877L)
((u'Phishing', '05-2019'), 6383827559227724320)
((u'Phishing', '06-2017'), 1000000000000000000)
((u'Phishing', '06-2018'), 835409548087648099125L)
((u'Phishing', '06-2019'), 8087272214976406262)
((u'Phishing', '07-2017'), 6315819990246015626844L)
((u'Phishing', '07-2018'), 91317597587664240011L)
((u'Phishing', '08-2017'), 697498484656474995606 8L)
((u'Phishing', '08-2018'), 26038246582858469476L)
((u'Phishing', '09-2017'), 349267458482069398470 8L)
((u'Phishing', '09-2018'), 27232229905500000000L)
((u'Phishing', '10-2017'), 1909578806305940952655L)
((u'Phishing', '10-2018'), 13457963543301563693L)
((u'Phishing', '11-2017'), 1931130818121393033801L)
((u'Phishing', '11-2018'), 65608480333514113439L)
((u'Phishing', '12-2017'), 1030068495042278537760L)
((u'Phishing', '12-2018'), 113564575456080000000L)
((u'Scamming', '01-2018'), 800153769349850718686L)
((u'Scamming', '01-2019'), 114339300553338156973L)
((u'Scamming', '02-2018'), 498402535182915198977L)
((u'Scamming', '02-2019'), 81074252993430518407L)
((u'Scamming', '03-2018'), 2787975026769754334083L)
((u'Scamming', '03-2019'), 145015667533719465609 1L)
((u'Scamming', '04-2018'), 2234444404319805847088L)
((u'Scamming', '04-2019'), 163228550055007043252L)
((u'Scamming', '05-2018'), 1863248926582830279055L)
((u'Scamming', '05-2019'), 183004992841313501198L)
((u'Scamming', '06-2017'), 98784101200000000000L)
((u'Scamming', '06-2018'), 1994605998015701010629L)
((u'Scamming', '06-2019'), 196490072655789755645L)
((u'Scamming', '07-2017'), 1238944359754270060501L)
((u'Scamming', '07-2018'), 2032760718001012530953L)
((u'Scamming', '08-2017'), 29950742870000000000L)
((u'Scamming', '08-2018'), 732794263202598956076L)
((u'Scamming', '09-2017'), 181698633896218700114L)
((u'Scamming', '09-2018'), 17802639887097456167342L)
((u'Scamming', '10-2017'), 1839392288663253070196L)
((u'Scamming', '10-2018'), 1621772957192355540937L)
((u'Scamming', '11-2017'), 2628927781457444576)
((u'Scamming', '11-2018'), 180491388841119222190L)
((u'Scamming', '12-2017'), 27509581908918747084L)
((u'Scamming', '12-2018'), 340194596436112270291L)
```

The Spark job below was used to obtain the most lucrative scam and its change over time.

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

def clean_transactions(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')
trans_filtered = transactions.filter(clean_transactions)
transactions_join = trans_filtered.map(lambda l: (l.split(',')[2] , (int(l.split(',')[6]), int(l.split(',')[3])))).persist()

scams = sc.textFile('/user/rkt31/scams.csv')
join_scams = scams.map(lambda f: (f.split(',')[0],f.split(',')[5]))
join_trans_scams = transactions_join.join(join_scams)


scam_category = join_trans_scams.map(lambda a: (a[1][1], a[1][0][1]))
scam_category_count = scam_category.reduceByKey(lambda a,b: (a+b)).sortByKey()
scam_category_count.saveAsTextFile('most_lucrative_scam')

time_analysis = join_trans_scams.map(lambda b: ((b[1][1], time.strftime("%m-%Y",time.gmtime(b[1][0][0]))), b[1][0][1]))
output = time_analysis.reduceByKey(lambda a,b: (a+b)).sortByKey()
output.saveAsTextFile('timeanalysis')
```

Clean the transactions dataset such that there are no missing fields

Load the transactions dataset. Filter the data and map the filtered data using to_address as key and block_timestamp & value as the value

Load scams dataset after converting it into .csv from .json

Map the scams data with address as the key and category of scams as value

Join the output from transactions and scams data using the address as key

Reduce by the key to get most lucrative and the variation with time

2.Comparative Evaluation Reimplement Part B in Spark (if your original was MRJob, or vice versa). How does it run in comparison? Keep in mind that to get representative results you will have to run the job multiple times, and report median/average results. Can you explain the reason for these results? What framework seems more appropriate for this task? (10/30)

	Time taken to run first time	Time taken to run second time	Average time taken to run
Spark Part B	8 mins , 36 sec	5 mins , 0 secs	408 secs =6.8mins
MR Job1 PartB	38 mins , 58 sec = 2338 secs	45 mins , 32 sec=2732 secs	2535secs = 42.25 mins
MR Job2 PartB			
MR Job3 PartB			

Spark seems more appropriate for this task because it takes less time to run compared to Map Reduce. Also, here were are using the output of job1 as an input to the job2 and output of job2 as input to job3.For performing such iterative tasks in-memory processing is required. Hadoop does not do in-memory processing, every time the output is saved in a disk and must be extracted from the disk for the next input. Hence, spark is appropriate for executing this job.

MapReduce for Job1

JOB ID :
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1575381276332_3326

```
from mrjob.job import MRJob
class PartB1(MRJob):


    def mapper (self, _,line):
        try:
            fields = line.split(',') #split the lines
            if len(fields) == 7: #to check if there are any missing fields
                toaddress = (fields[2]) #extracting to_address from field2
                value = int(fields[3]) #extracting value from field3
                yield(toaddress,value) #yielding the mapper results

        except:
            pass

    def combiner(self, toaddress, value):#to sum all the values
        yield(toaddress,sum(value))


    def reducer(self, toaddress, value):#to sum all the values
        yield(toaddress,sum(value))

if __name__ == '__main__':
    PartB1.run()
```

Submitted By,

Tampu Ravi Kumar