Name : Priyank Mishra

Student Number : 190632629

Module : ECS765P - BIG DATA PROCESSING - 2019/20

COURSEWORK: ETHEREUM ANALYSIS (20%)

PART A. TIME ANALYSIS (30%)

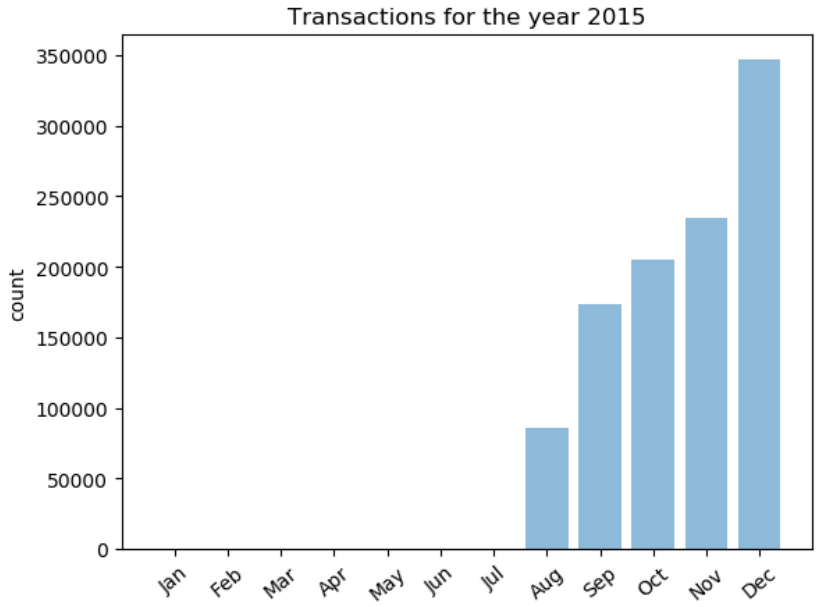
Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

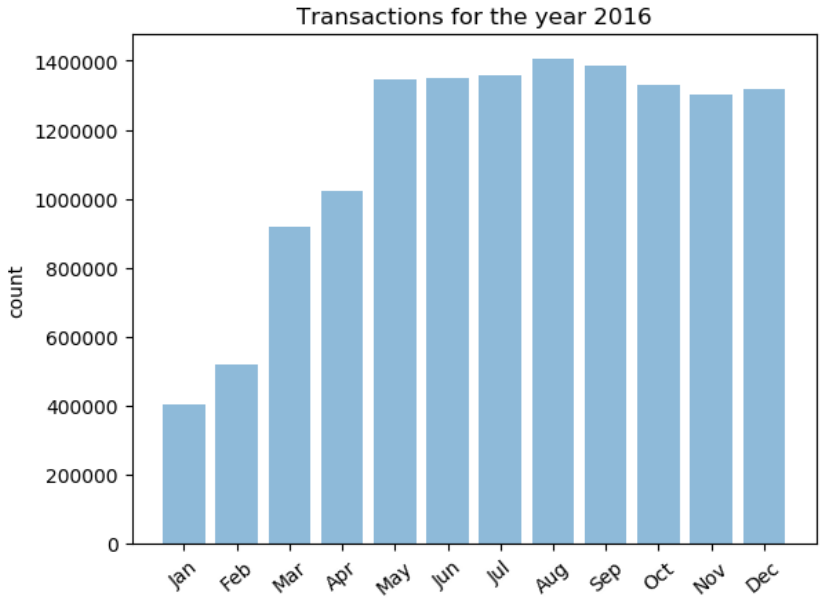
Note: As the dataset spans multiple years and you are aggregating together all transactions in the same month, make sure to include the year in your analysis.

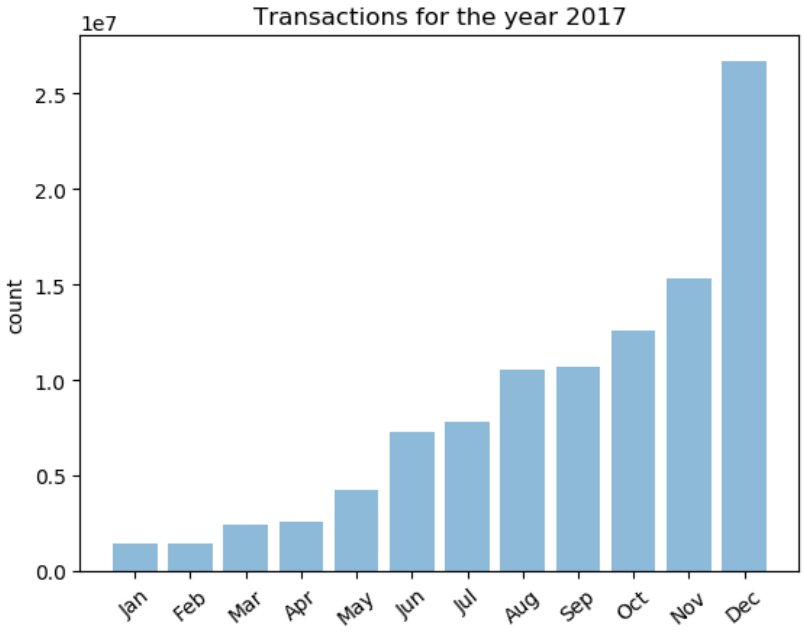
Note: Once the raw results have been processed within Hadoop/Spark you may create your bar plot in any software of your choice (excel, python, R, etc.)

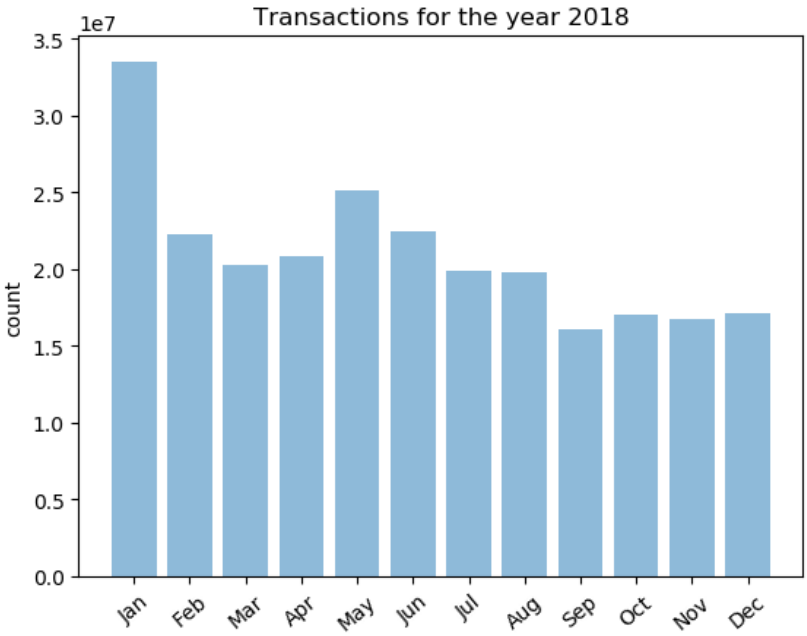
Answer :

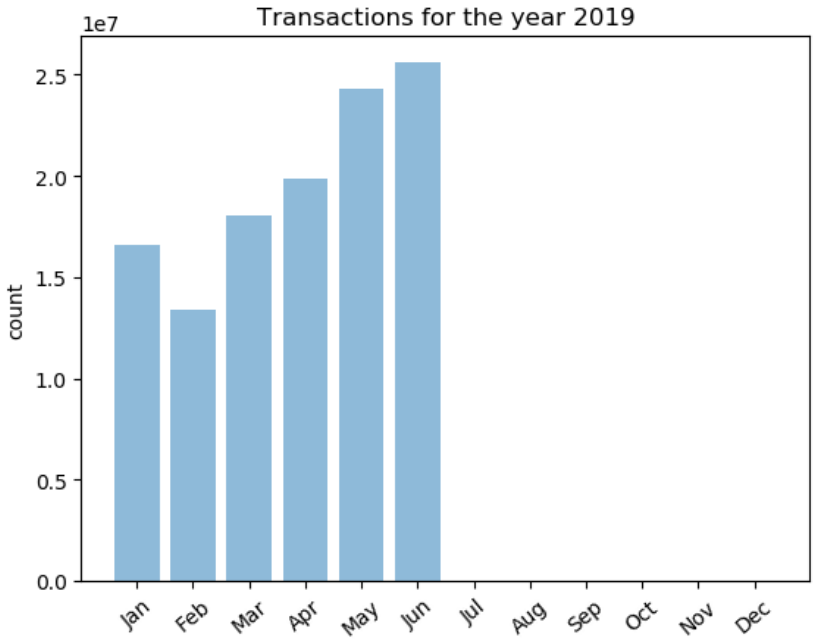
The bar plots showing all the total number of transactions in a month for each year are as shown below :



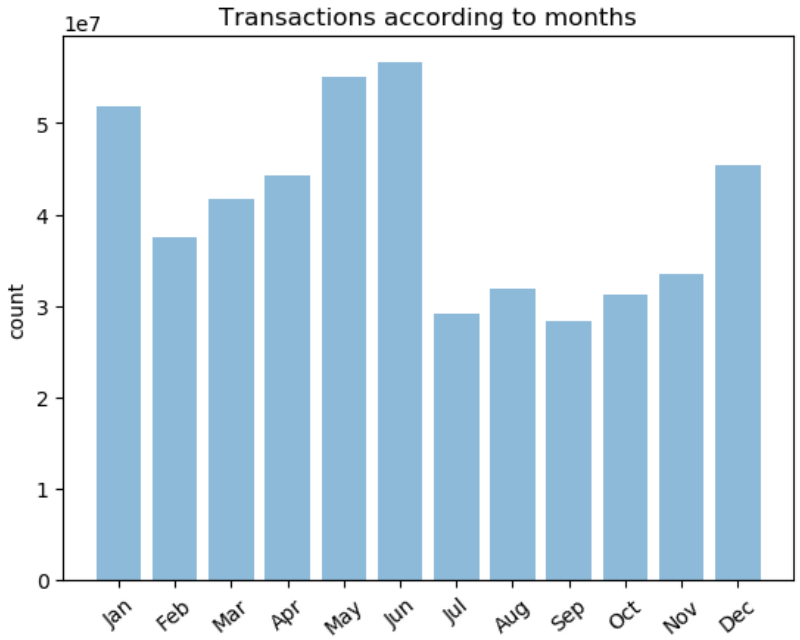

Transactions for the year 2016
count
1400000
1200000
1000000
800000
600000
400000
200000
0
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec


Transactions for the year 2017
1e7
count
2.5
2.0
1.5
1.0
0.5
0.0
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec





We can also plot the graph for all the years combined , month wise total number of transactions , which looks like this :



Code implemented in the file " timeanalysis_with_year.py"

Job Id :
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3973/

Explanation of the code implemented :

1. In the mapper we first split our incoming data line using comma as the separator as our "transactions" data is in CSV(comma separated values) format . Then we check if we have 7 elements in the resulting list or not , so that we can ignore any malformed lines. Now we take the timestamp stored in the column with index 6 , and convert it into integer and store it int the variable "time_epoch" . Then we apply the function " time.gmtime()" on it , which returns a named tuple with all the information like year , month , day , etc. Then we use the function "time.strftime()" and use "%m" to get the year , and "%Y" to get the year. We then yield the key-value pair as , key = (month,year) , value = (1) .
   Lines from code :
   def mapper(self, _, line):

```
fields=line.split(",")
try:
   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))
            yield((month,year),1)
        except:
            pass
```

2. We can use a simple combiner , identical to the reducer to sum all the values for a single key. This will emit partial counts, as the combiner function is applied on the data from the mapper before shuffle and sort stage. This helps in speeding up our program , as there will be lesser data to send across the network in the shuffle & sort stage.
   Lines from code :

```
def combiner(self, key, counts):
    yield(key,sum(counts))
```

3. The data emitted from the combiner, after going through the shuffle and sort stage , comes to the reducer. Which will yield , key = (month, year) , and value = (total number of transactions occurred) as the final result .
   Lines from code :

```
def reducer(self, monthyear, counts):
    yield(monthyear,sum(counts))
```

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.

Answer :

Code is implemented in the file "aggregate_tran.py"

Job ID :

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3578/

Explanation of the code implemented :

1. Since we know that the "transactions" data set is a CSV file (comma separate values), In the mapper the first step is to split the line that the Hadoop feeds using comma as the separator. Then we will check if there are 7 elements in the resulting list or not, if there are not exactly 7 elements then it means that the line was malformed and is unusable for our purpose and we can ignore it. For the correctly formed lines we will emit from the mapper "to_address"(column index 2) as the key and "value" (column index 3) as value.

   Lines from code :

```
def mapper(self, _, line):
        fields=line.split(",")
        try:
            if (len(fields)==7):
            to_address = fields[2]
            value = float(fields[3])
             yield(to_address,value)
        except:
            pass
```

2. Here we can use a combiner identical to the reducer , which will perform summation of all the values corresponding to a particular key. Using combiner will help to reduce the amount of data that will need to cross the network. This will make the job run faster as there will be lesser data for the shuffle & sort phase .

   Lines from code:

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

yield(address,sum(value))

3. The reducer will finally aggregate all the values for a particular key, and thus give as output the total wei value corresponding to a key of "to_address" .

   Lines from code :

   def reducer(self, address, value):

   yield(address,sum(value))

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.

Answer :

Code is implemented in the file " repartition_join.py"

Job ID :
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3692/

Explanation of the code :

1. In this code we will be using the output of the previous program (aggregate_tran.py) . This data will be in the tab separated value (tsv) format. The data for "contracts" is in comma separated value (csv) format . This format difference is beneficial to us , because we can use it to identify which data set the data is coming from . Again, we will want to check whether there are required number of elements in the list after splitting line or not , so that we can ignore the malformed line. If the data comes from the aggregate data set key = (to_address),

and value = (aggregate wei values , 1). If the data comes from “contracts” the mapper will emit key = (address) ,and value =(block number,2). This extra “1” and “2” that we have used in the values of the key-value pairs will help us to identify the source of the data when we receive it in the reducer .

Lines from code :

```
def mapper(self, _, line):
        try:
            if len(line.split('\t'))==2: ## this is aggregate transactions stored in the "out" folder
                fields = line.split('\t')
                join_key = str(fields[0])[1:-1]  ## to_address
                join_value = float(fields[1]) ## wei values
                yield (join_key, (join_value,1))

            elif len(line.split(','))==5:
                fields = line.split(',')
                join_key = str(fields[0])
                join_value = float(fields[3])
                yield (join_key,(join_value,2))
        except:
            pass
```

2. In the reducer we identify where the data is coming from by checking if the values of the key-value pairs contain “1” or “2” . So when a key-value pair reach the reducer , we run a loop for all the data that is present in the “value” part and accordingly copy the data into separate variables . we also use the variable “filter1” and “filter2” to check if atleast once we have received the data from both the data sets for a single key . The new values emitted contain the key = (address, block number) , value = (total wei value)

Lines from code :

```
    def reducer(self, address, values):
        try:
            contract_info = 0
            wei_value = 0.0
            filter1 = 0
            filter2 = 0

            for value in values:

                if value[1] == 1:
                    wei_value = value[0]
                    filter1 = 1
                elif value[1] == 2:
                    filter2 = 1
                    contract_info = value[0]

             if filter1 == 1 and wei_value > 0 and filter2 == 1:
                    yield ((address,contract_info),wei_value)
        except:
            pass
```

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.

Answer :

Code is implemented in the file "topten_eth.py"

Job ID :
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3704/

Explanation of the code :

1. Here we will be using the output of the previous program (repartition_join.py), to find the top ten values on the basis of total wei value . Since our previous output is a Tab Separated value (tsv) we will use "\t" as the separator in the "split()" function. After splitting , we emit key = (None) , and value = (col_1,wei_value) . Since we have kept a constant value (rather None value) all the key-value pairs will go to the same reducer.

   Lines from code :

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

    try:

        fields=line.split("\t")
        col_1 = fields[0]
        wei_value = float(fields[1])
        yield(None,(col_1,wei_value))

    except:
        pass
```

2. We can use a combiner to perform sorting on the partial results, and emit only top ten values in the partial results to reduce the load.

   Lines from code :

```
def combiner(self, key , value):
    vals = sorted(value, reverse=True, key=lambda l:l[1])
```

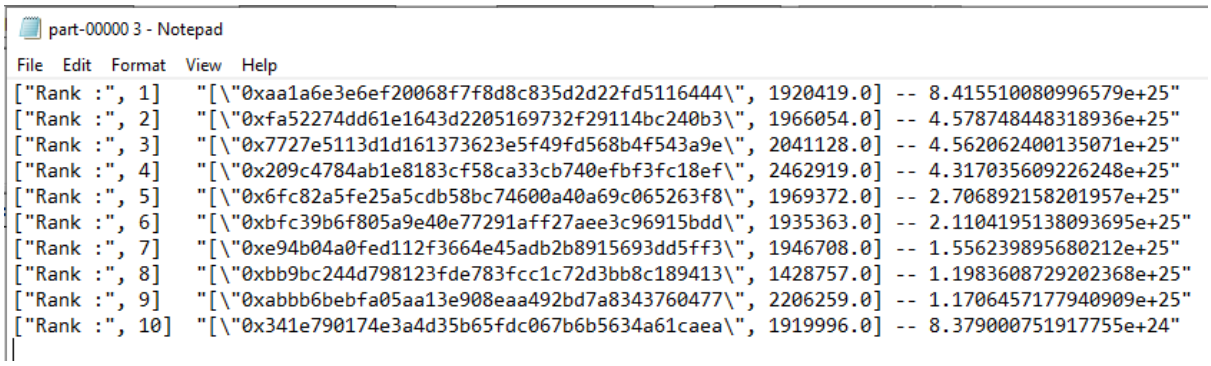
```
        vals = vals[0:10]
        for i in vals:
            yield(None,i)
```

3. In the reducer we will again use the “sorted” function to sort the data and then we will just select the top ten of the results. Then we run a for loop to yield the data one by one .

   Lines from code :

```
    def reducer(self, key, value):
        vals = sorted(value, reverse=True, key=lambda l:l[1])
        vals = vals[0:10]
        cnt = 0
        for i in vals:
            cnt +=1
            x='{} -- {}'.format(i[0],i[1])
            yield(("Rank :",cnt),x)
```

The final Output of this program looks like this :



PART C. DATA EXPLORATION (30%)

Question :

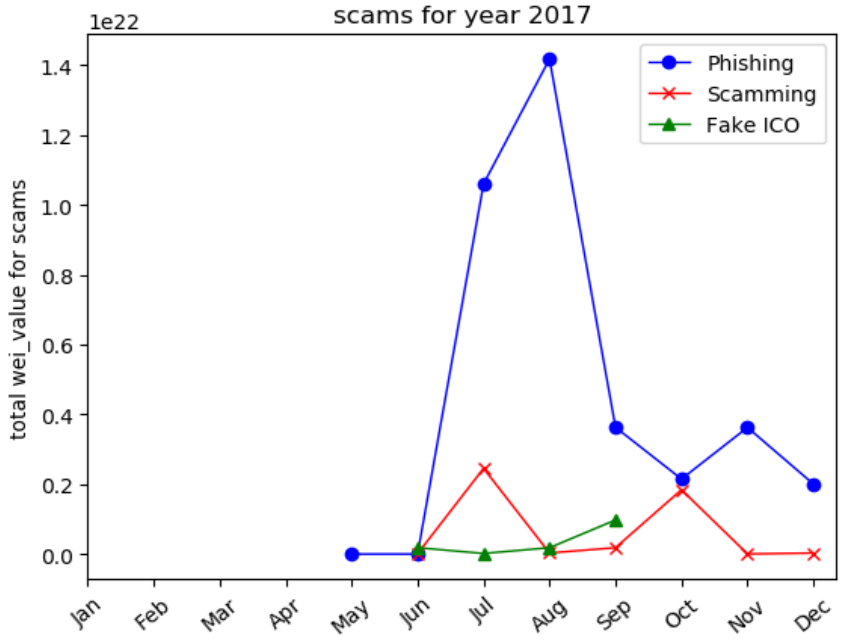
SCAM ANALYSIS

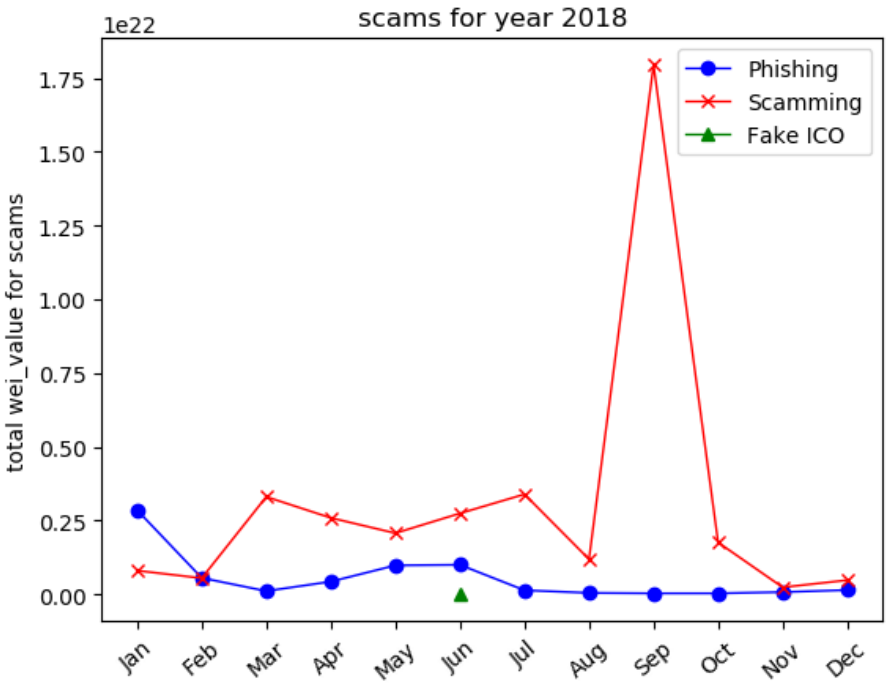
1. **Popular Scams: Utilising the provided scam dataset, what is the most lucrative form of scam? How does this change throughout time, and does this correlate with certain known scams going offline/inactive? (20/30)**

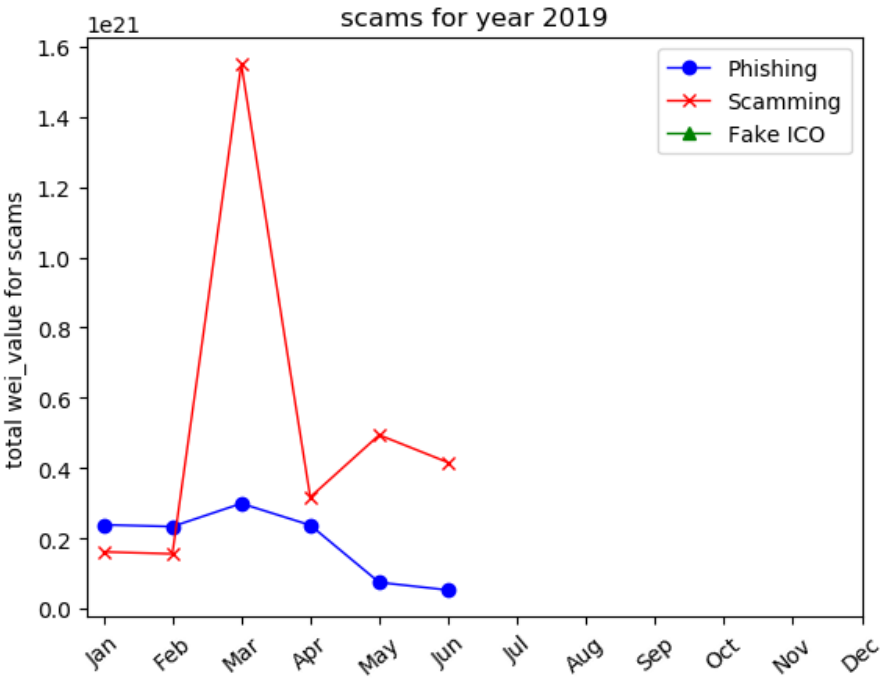
Answer :

We can conclude that the most lucrative form of scam is “Scamming” which happened 1747 times . We can say this is the most lucrative as it is happening a lot more times as compared to the other scams , so it will be more profitable . If we use the total wei values involved in these transactions even then we see that the “Scamming” is the most lucrative category of scam.

To see how the scams change through time we can plot the graph of the data generated from executing the code “**scam_tran_replication.py”** . Here we have shown 3 graphs for the three years (2017, 2018, 2019) for the three most popular scams , showing how big the scams were(in terms of total wei value) in different months.







we can see that these correspond with data in scams.json when a big scam goes offline there is drop in the graph.

If we search for the scams related to the peaks shown in the graphs we are able to find articles dated to the corresponding months where a lot of ether has been stolen.

For example in the graph for scams in year 2017 , we see there is a spike for the August month, and searching this on net we see there are some articles for that month mentioning a huge amount lost in scam(attaching the screenshot below): (link of the article : https://fortune.com/2017/08/28/ethereum-cryptocurrency-stolen-bitcoin/)

Link : https://www.pymnts.com/cryptocurrency/2018/ico-investors-exit-scams-phishing-attempts-bitcoin-ethereum/

Code executed to calculate the most lucrative scams "scam_top_ten.py"

Using the Scams.json file

Job Id and link :

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_2151/

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_2153/

Since we are using two steps in the code (two Map Reduce jobs) we get two separate Job IDs.

Code executed to generate the data of how the different types of scam vary with time : scam_tran_replication.py

Job ID and Link :

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_2221/

The data generated in this program was then used to plot graphs using python.

Explanation of the code in "scam_top_ten.py":

1. In this implementation we will use two mappers and reducers , to do this we use MRStep .
   Line from code :

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

   This enables us to write two map-reduce jobs in a single code

2. All the data present in the file "scams.json" is present as a single line. So when we execute our first mappers , Hadoop loads the entire data as a single line in the "line" parameter of the mapper "mapper1" . Then we load this data using "json.loads()" function into the variable "data" . The json data is basically present as key-value pairs and to get to our main data we can write " x = list(data['result'].values()) " which will create a dictionary " x " from where we will be able to retrieve our data using keys . Then we iterate over the elements of the " x " and yield values for the key "category" and 1 .

   Lines from code :

   ```
   def mapper1(self,_,line):
       try:
           data=json.loads(line)
           x = list(data['result'].values())
   ```

```
            for i in x :
                yield(i["category"],1)

        except:
            pass
```

3. All the key-value pairs emitted from the “mapper1” go through the shuffle & sort phase which brings the values having the same keys to the same reducer. Then in the “reducer1” we perform summation of the values received (which will be a list of 1s) for the keys . This will give us the total number of times a particular scam has happened . This reducer then emits the key-value pairs , which has category of scam as Key and the total number of occurrences as value.

   Line from code : def reducer1(self,category,value):

   ```
   yield(None,(category,sum(value)))
   ```

4. The “mapper2” is an identity mapper . It will not change the data , it will just emit the key value pairs .

5. The second reducer being used here “reducer2” will sort the values , and it will pick the top ten categories of scams .

   Line from code : def reducer2(self,key,value):

   ```
   vals = sorted(value, reverse=True, key=lambda l:l[1])
   vals = vals[0:10]
   for i in vals:
       yield(None,i)
   ```

On Implementing this we find that actually there are only 4 categories of scams , and the output looks like this :

null ["Scamming", 1747]

null ["Phishing", 587]

null ["Fake ICO", 5]

null ["Scam", 1]

Explanation of the code in "scam_tran_replication.py":

1. Here we use the mapper initialization function "mapper_init" which runs once before the mapper starts executing . We use this to Load our data "scams.json" into each mapper , so that we will be able to perform the replication join on the scams data and the transactions data . We are doing this because the Scams data does not have the timestamp, which is required to see how the scams haves changed over time .
   Line from code :

```
def mapper_init(self):
    with open('scams.json') as f:
        for line in f:
            data=json.loads(line)
            self.x = list(data['result'].values())
```

2. Now since we are doing the "replication " join it will happen in the mapper (and it will be much faster too as it doesn't not involve shuffle & sort step . We will basically join the two data sets by matching the values in "addresses" (from the scams.json) and the "to_address" (from the transactions data). And then we will yield the key as a combination of "category , month, year" and the value will contain the address and the wei_value.
   Lines from code :

```
def mapper(self,_,line):  ## doing Replication Join here
    try:
      if len(line.split(','))==7:
```

```
            fields = line.split(',')
            address = fields[2]
            for i in self.x:
               for j in i["addresses"]:
                  if j == address:
                     time_epoch=int(fields[6])
                     month = time.strftime("%m",time.gmtime(time_epoch))
                     year = time.strftime("%Y",time.gmtime(time_epoch))
                     wei_value = float(fields[3])
                     yield((i["category"],month,year),(address,wei_value))



      except:
         pass
```

3. The Reducer will calculate the total wei value for a particular key and emit it as “total_wei” . So we will final get the total wei value for a particular category of scam in a particular month of an year.

   Lines from code :

```
            def reducer(self,key,value):
                  total_wei = 0.0
                  for i in value:
                        total_wei = total_wei + i[1]


                  yield(key,total_wei)
```

Question :

MISCELLANEOUS ANALYSIS

3. 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)

Answer :

Name of the python code file is "sparkpartb.py"

Explanation of the code implemented :

1. First of all we define two functions namely "clean_transactions_func(tranc)" and "clean_contracts_func(cont)" **.**

   These functions are used to filter the data , so that we don't include any malformed line. One simple way to achieve this is to check how many elements are present when we split a line. There should be 7 elements (length = 7) when we split a line of the "transactions" input . And there should be 5 elements in the split of the line of "contracts" input.

2. In the code , "sc" is the spark context object which will be used to create the initial RDD

   Line from Code : sc = pyspark.SparkContext()

   This will be used to create our first RDD "tranc" which will have the data from the files stored in the "transactions " folder .

   Line from Code : tranc = sc.textFile("/data/ethereum/transactions")

3. Then we filter the "transactions" records using the functions we have defined in the beginning so that we remove the malformed lines .

4. Then we use the "map" transformation that passes each data set element through a function and returns the a new RDD which represents the results. The function retrieves only two columns , column for the "to_address" (which is at index 2) and the column for the Wei "Value" (which is at index 3) . And the resulting RDD will have only these two columns .

5. Then we use the transformation " reduceByKey " which will generate another RDD and uses a function which should be associative and commutative in nature which will be applied to the "values" of the key value pairs . The new RDD will contain the "to_address" as the keys , and will have the total wei value for the particular key .

   Line from the code : job1_out = tranc_values.reduceByKey(lambda a,b:(a+b))
   job1_out_join=job1_out.map(lambda f:(f[0], f[1]))

   So this gives the "aggregate transactions" which corresponds to the "Part B , Job 1" of the map reduce programs implemented .

6. Now we load the "contracts" data , on which we will perform some transformations .
   Line from code : contracts = sc.textFile("/data/ethereum/contracts")

7. Similar to the "Transactions" data , we will apply the filter function we had defeined earlier to remove any malformed line . A proper line in the "contracts" file should have 5 elements after the split function is applied .
   Line from code : contracts_filtered = contracts.filter(clean_contracts_func)

8. Now we apply the "map" transformation on the filtered "contracts" data and maintain only two columns , column for the "to_address"(index 0) and the column for the "block number" (index 3), in our new RDD .
   Line from code : contracts_join = contracts_filtered.map(lambda f:
   (f.split(',')[0],f.split(',')[3]))

9. Now we apply the "join" transformation to join the data that we retrieved as key-value pairs from the "transactions" data (stored in job1_out_join) with the data that we retrieved from the "contracts" data (stored in contracts_join). This join will use "to_address" as the joining key (primary key) , and so only those rows will be maintained in our new RDD for which the same value of "to_address" is present in both the data sets .
   Line from code : job2_out = job1_out_join.join(contracts_join)

   So here we have completed the " Part B , Job 2 " tasks .

10. Now we have the joined data in "job2_out" , from which we just need to take out the top ten rows according to the aggregate wei values. For this spark has a an Action "takeOrdered" .

Line form code : top_ten_out=job2_out.takeOrdered(10, key = lambda x:-x[1][0])

Where “10” represents that only 10 values are needed and the minus sign in the lambda ensures the value are sorted in Decreasing order.

Then we can just iterate through the values and print them .
Line from code : for record in top_ten_out:
print("{}: {}".format(record[0],record[1][0]))

out put of the program is (arranged in decreasing order of the total Wei value associated to a to_address):

to_address	Total Wei value
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444:	84155100809965865822726776
0xfa52274dd61e1643d2205169732f29114bc240b3:	45787484483189352986478805
0x7727e5113d1d161373623e5f49fd568b4f543a9e:	45620624001350712557268573
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef:	43170356092262468919298969
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8:	27068921582019542499882877
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd:	21104195138093660050000000
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3:	15562398956802112254719409
0xbb9bc244d798123fde783fcc1c72d3bb8c189413:	11983608729202893846818681
0xabbb6bebfa05aa13e908eaa492bd7a8343760477:	11706457177940895521770404
0x341e790174e3a4d35b65fdc067b6b5634a61caea:	837900075191775562405750 0

Job IDs for the Spark Program ran for the Comparative Analysis question :

S.No.	Job ID	Elapsed Time (in seconds)
1	application_1575381276332_4123	150
2	application_1575381276332_4127	174
3	application_1575381276332_4134	97
4	application_1575381276332_4137	96

So we can say average time it takes for the spark job to run is : 129.25 seconds

Executing all the Map Reduce codes for the Part B , several times , so that we can calculate the average time required to get the final results .

S. No.	Job name	Job ID	Elapsed time (in seconds)	Total time (in seconds)
1	Job 1	application_1575381276332_4064	373	567
	Job 2	application_1575381276332_4073	168	
	Job 3	application_1575381276332_4082	26	
2	Job 1	application_1575381276332_4086	287	428
	Job 2	application_1575381276332_4091	125	
	Job 3	application_1575381276332_4094	16	
3	Job 1	application_1575381276332_4100	357	488
	Job 2	application_1575381276332_4103	115	
	Job 3	application_1575381276332_4105	16	
4	Job 1	application_1575381276332_4109	280	410
	Job 2	application_1575381276332_4110	114	
	Job 3	application_1575381276332_4111	16	

Therefore Average time required to get the final result in Hadoop is : 473.25 seconds

So we see that in comparison to Hadoop , Spark job takes a lot lesser time to run .

Also the implementation of the jobs is quite easy , as Spark has some predefined Transformations and Actions that are ready to use on the data. Unlike Hadoop , where we had to implement the code for join or to find the top ten values .

Also here we have the ability to persist the RDDs in memory so that any new Transformation can be easily applied on it, without the need to calculate the results again or retrieving it from the HDFS which would consume time.

So we can conclude , that Spark is more appropriate for these type of tasks.

Also we can observe that , the execution time changes each time we run our job no matter whether it is Hadoop or spark . This happens because at the time of execution of our job , the number of resources available might be different, as other users are also executing their jobs on the cluster. Due to this at different times the cluster might be processing a different number of jobs and therefore different amount of resources will be available, and this will impact the performance.