PART- A. TIME ANALYSIS

Code for Time Analysis:

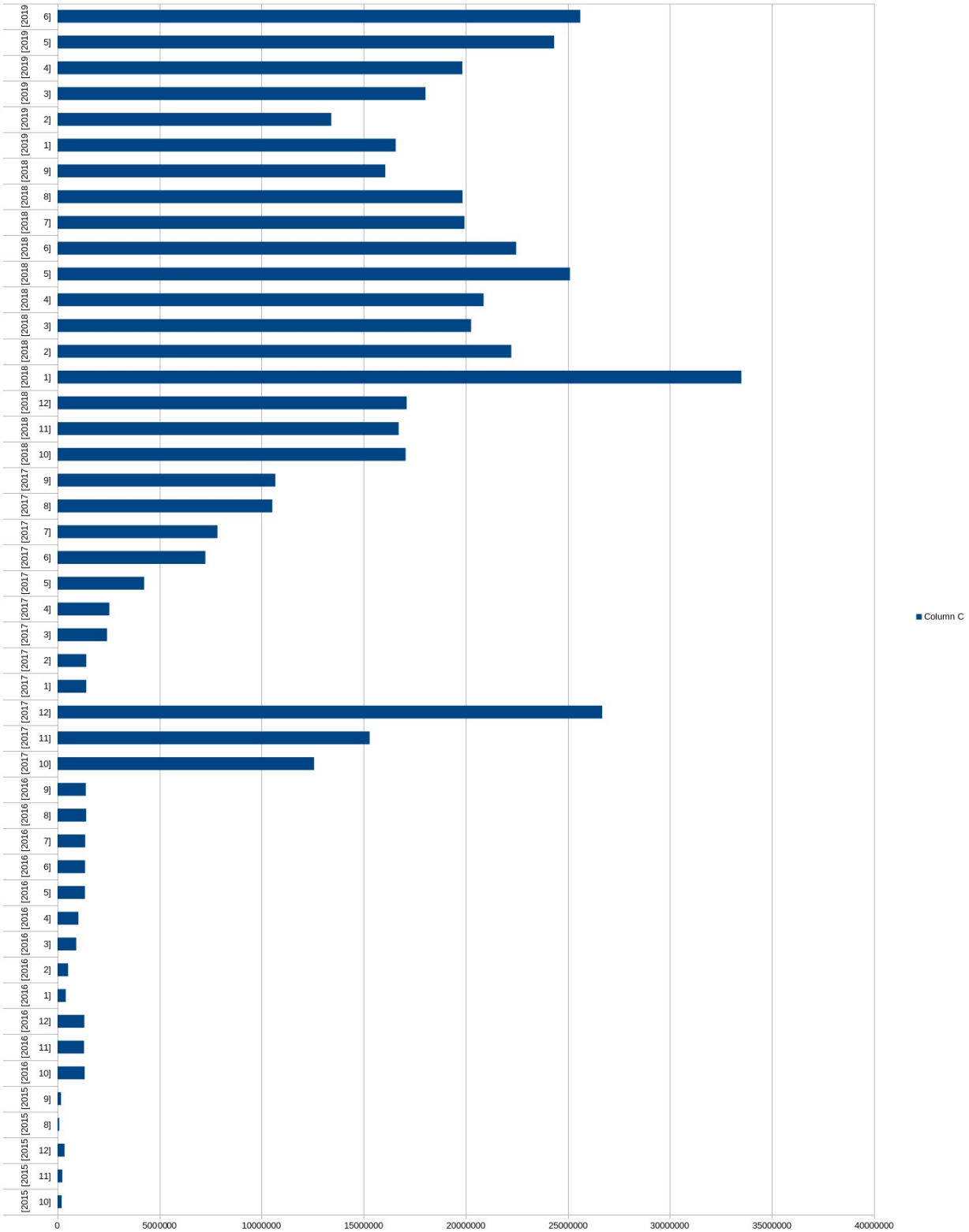
```
import time
from mrjob.job import MRJob

class PartA(MRJob):

    def mapper(self, _, line):
        fields = line.split(',')
        try:
            t = time.gmtime(int(fields[6]))
            year = t.tm_year
            month = t.tm_mon
            yield((year, month), 1)
        except:
            pass

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

    def reducer(self, tm, count):
        yield(tm, sum(count))

if __name__ == '__main__':
    PartA.JOBCONF = {'mapreduce.job.reduces': '1'}
    PartA.run()
```

Explanation: In the above code, we are aggregation all the transactions occurring in different months and year. The mapper is taking the data from transactions and splitting it into lines to extract months and years. The epoch time is getting converted into gmt time from field 6, which will give an integer value. The combiner will give the sum values for each key emitted by mapper. Finally, the reducer will give the total sum of all transactions occurring every month and year.

The bar graph shows that in the first month of year 2018, there was highest number of transactions.

The Job ID for Part A is:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574935476688_0352/


[2019 6]
[2019 5]
[2019 4]
[2019 3]
[2019 2]
[2019 1]
[2018 9]
[2018 8]
[2018 7]
[2018 6]
[2018 5]
[2018 4]
[2018 3]
[2018 2]
[2018 1]
[2018 12]
[2018 11]
[2018 10]
[2017 9]
[2017 8]
[2017 7]
[2017 6]
[2017 5]
[2017 4]
[2017 3]
[2017 2]
[2017 1]
[2017 12]
[2017 11]
[2017 10]
[2016 9]
[2016 8]
[2016 7]
[2016 6]
[2016 5]
[2016 4]
[2016 3]
[2016 2]
[2016 1]
[2016 12]
[2016 11]
[2016 10]
[2015 9]
[2015 8]
[2015 12]
[2015 11]
[2015 10]
0
5000000
10000000
15000000
20000000
25000000
30000000
35000000
40000000
Column C

PART- B. TOP TEN MOST POPULAR SERVICES

JOB 1- INITIAL AGGREGATION

Code for JOB 1:

```
from mrjob.job import MRJob

class PartBA(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split(',')
            addr = fields[2]
            value = int(fields[3])

            if value == 0:
                pass
            else:
                yield(addr, value)
        except:
            pass
    def combiner(self, addr, value):
        yield(addr, sum(value))

    def reducer(self, addr, value):
        yield(addr, sum(value))

if __name__ == '__main__':
    #PartBA.JOBCONF = {'mapreduce.job.reduces': '1'}
    PartBA.run()
```

Explanation: In the above code, we are aggregating transactions. In the mapper, we are finding the address from field 2 and their values from field 3. The mapper will emit the key as output. The combiner will give the sum of values for the keys emitted by mapper. The reducer will give the total of all values for addresses in the to_address field.

The Job ID for Part B- JOB 1 is:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574935476688_0391/

JOB 2- JOINING TRANSACTIONS/CONNTRACTA AND FILTERING

Code for JOB 2:

```
from mrjob.job import MRJob

class PartBB(MRJob):

    def mapper(self, _, line):
        try:
            if len(line.split(',')) == 5:
                fields = line.split(',')
                join_key1 = fields[0]
                join_value = fields[3]
                yield(join_key1, (join_value, 1))

            if len(line.split('\t')) == 2:
                fields = line.split('\t')
                join_key2 = fields[0]
                join_key2 = join_key2[1:-1]
                join_value = int(fields[1])
                yield(join_key2, (join_value, 2))


        except:
            pass

    #def combiner(self, addr, value):
        #yield(addr, sum(value))

    def reducer(self, addr, values):
        block = None
        amt = None


        for value in values:
            if value[1] == 1:
                block = value[0]

            if value[1] == 2:
                amt = value[0]

        if not block is None and not amt is None:
```

```
            yield((addr, block), amt)
if __name__ == '__main__':
    #PartBB.JOBCONF = {'mapreduce.job.reduces': '1'}
    PartBB.run()
```

Explanation: In the above code, we are performing repartition join between the aggregate transactions and contracts. The mapper will take the input from the output of Job1 and the address field of contracts. The mapper will emit two different keys as per splitting the lines and it will join both. The reducer will give the address, block and amount.

The Job ID for Part B- JOB 2 is:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_0726/

JOB 3- TOP TEN

Code for JOB 3:

```
from mrjob.job import MRJob

class PartBC(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split()
            addr = fields[0][2:-2]
            block = fields[1][1:-2]
            addr = addr + ' - ' + block
            value = int(fields[2])
            yield(None, (addr, value))

        except:
            pass

    #def combiner(self, addr, value):
        #yield(addr, sum(value))

    def reducer(self, _, values):
        values = sorted(values, reverse=True, key=lambda l: l[1])
        values = values[:10]
        for value in values:
            addr = value[0]
            val = value[1]
```

```
        yield(addr, val)

if __name__ == '__main__':
    #PartTwo.JOBCONF = {'mapreduce.job.reduces': '1'}
    PartBC.run()
```

Explanation: In the above code, the mapper will take the filtered address aggregates as input and the reducer will sort them and provide the top 10 values.

The Job ID for Part B- JOB 3 is:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_1223/

PART- C. DATA EXPLORATION

SCAM ANALYSIS

Code to find the top 10 categories of SCAM:

```
class popular_scams(MRJob):

    sector_table = {}

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


    def mapper1(self,_,line):
        try:
            data=json.loads(line)
            a = list(data['result'].values())
            for i in a :
                yield(i["category"],1)

        except:
            pass


    def reducer1(self,category,value):
```

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


    def mapper2(self,key,value):
        yield(key,value)

    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)

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

Explanation: In the above code, we are using two mappers and reducers to find the top 10 categories of scam. The first mapper will retrieve the data from the json file as keys. These keys will go to the reducer1 passing through shuffle and sort. From reducer1, we will the number of scams happened. Reducer1 will emit the key value pairs and pass it to the mapper2. It will send the values to reducer2. This reducer2 will give the top categories of scam and the output will be

```
null    ["Scamming", 1747]
null    ["Phishing", 587]
null    ["Fake ICO", 5]
null    ["Scam", 1]
```

The Job id for both the mappers and reducers are:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3552/
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3553/

Code for scams which changes throughout time:

```
class popular_scams(MRJob):

    z = {}

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

```
    def mapper(self,_,line):
        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))
                            w_value = float(fields[3])
                            yield((i["category"],month,year),(address,w_value))

        except:
            pass

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

if __name__ =='__main__':
    popular_scams.JOBCONF = { 'mapreduce.job.reduces': '2' }
    popular_scams.run()
```

Explanation: In the above code, we are defining how scams changes with time, for which we are matching the addresses of scams.json with the addresses in the transaction data. The mapper will give the key consisting of category, month and year. The reducer will calculate the total (total_w) for a particular key. So, finally we will get to know the value for every category in a month and a year.

The Job id for the above code is:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_3668/

MISCELLANEOUS ANALYSIS- 3. COMPARATIVE EVALUATION

Code of Part B in Spark for comparative analysis is:

```
import pyspark

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

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

sc = pyspark.SparkContext()

transactions = sc.textFile("/data/ethereum/transactions")
val_transactions = transactions.filter(val_trans)
mapper_transactions = val_transactions.map(lambda l: (l.split(',')[2], int(l.split(',')[3])))
aggregate_transactions = mapper_transactions.reduceByKey(lambda a, b: a+b)


contracts = sc.textFile("/data/ethereum/contracts")
val_contracts = contracts.filter(val_cont)
mapper_contracts = val_contracts.map(lambda l: (l.split(',')[0], None))

join1 = aggregate_transactions.join(mapper_contracts)

top10 = join1.takeOrdered(10, key = lambda k: -k[1][0])

for addr in top10:
    print('{} - {}'.format(addr[0], addr[1][0]))
```

Explanation:-

1. In the above code, we are defining the elements first by splitting the lines of transactions and contracts. So, there are 7 elements from transactions and 5 elements from contracts. We are using 'sc' which is Spark Context which is used to create the initial RDD.

2. From the transaction part above, we will get the aggregate transactions which we got in Job 1 of part B. Once this is done, we will perform operations on the contract part and we will join the aggregate transactions with the data we got from the contracts. After joining, we will be able to complete Job 2 of part B.

3. We will then find the top 10 values.

4. Here, we saw that as compared to hadoop which we did in Part B, the implementation of code in spark is simpler than the hadoop. Also, it is time efficient. Here, in spark, everything is stored in RDD, so there is no need to write the code again to bring the changes, we can directly retrieve it from RDD. Hence, Spark is more useful for such type of problems.

The Job id for the above code of comparative evaluation is:

application_1575381276332_1537