Big Data Processing Coursework

PART A. TIME ANALYSIS (30%)

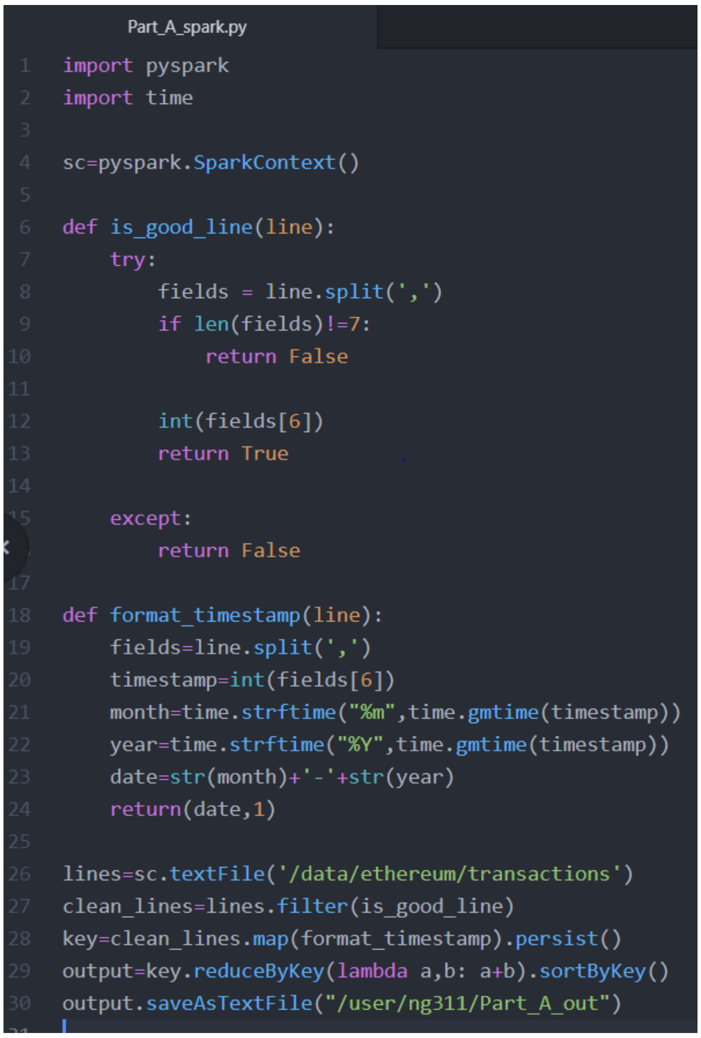
Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.
Note: As the dataset spans multiple years and you are aggregating together all transactions in the same month, make sure to include the year in your analysis.
Note: Once the raw results have been processed within Hadoop/Spark you may create your bar plot in any software of your choice (excel, python, R, etc.)

Approach:

In order to calculate the number of transactions occurring every month the **block_timestamp** field in **transactions** dataset is used. The aggregate value(sum) of all the timestamp values per month returns the total number of transactions occurred.

Code:

The spark code to create the plot is shown below. Program name is Part_A_spark.py



Part_A_spark.py

```python
import pyspark
import time

sc=pyspark.SparkContext()

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

        int(fields[6])
        return True

    except:
        return False

def format_timestamp(line):
    fields=line.split(',')
    timestamp=int(fields[6])
    month=time.strftime("%m",time.gmtime(timestamp))
    year=time.strftime("%Y",time.gmtime(timestamp))
    date=str(month)+'-'+str(year)
    return(date,1)

lines=sc.textFile('/data/ethereum/transactions')
clean_lines=lines.filter(is_good_line)
key=clean_lines.map(format_timestamp).persist()
output=key.reduceByKey(lambda a,b: a+b).sortByKey()
output.saveAsTextFile("/user/ng311/Part_A_out")
```

Code Explanation:

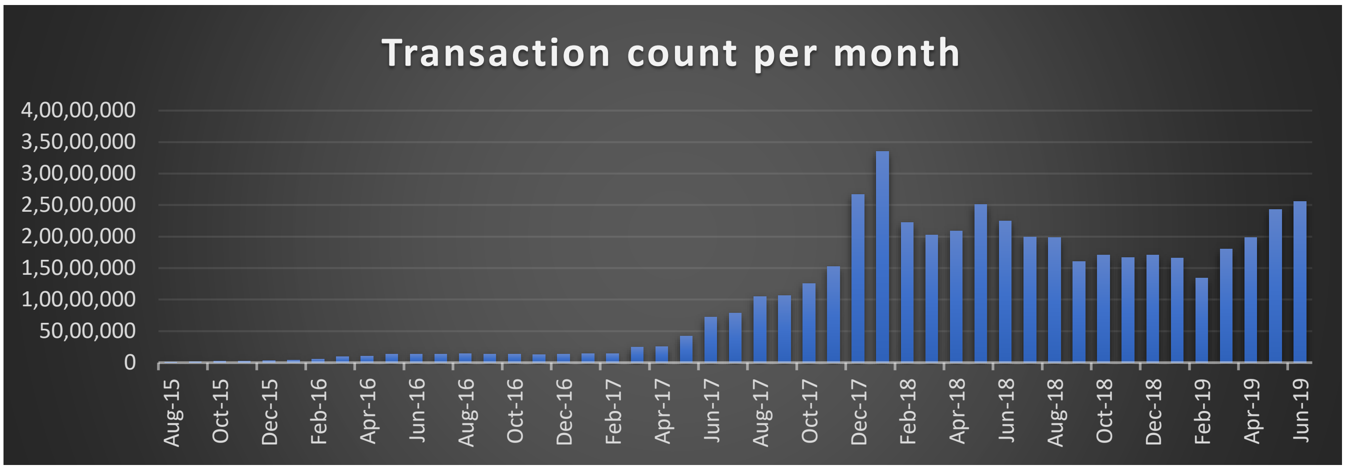
- Libraries **pyspark** and **time** has been imported to execute spark's inbuilt functions and format timestamp.
- **Is_good_line** function has been defined to filter out any bad lines while reading the **transactions file.** The lines are split by comma and is checked whether it contains only 7 fields. Also **block_timestamp** field is defined as **int** to process the timestamp values.
- **format_timestamp** function is defined to convert the unix timestamp values to Gregorian format. The **time** library is used to convert the timestamp values. **time.gmtime** converts the unix timestamp to the desired format. Here we convert it into MM-YY format.
- lines=sc.textFile('/data/ethereum/transactions') reads the transactions dataset using spark context sc.
- clean_lines=lines.filter(is_good_line) filters out the bad lines.
- key=clean_lines.map(format_timestamp).persist() maps the converted timestamp values.
- output=key.reduceByKey(lambda a,b: a+b).sortByKey() performs the aggregation on the timestamp value which returns the total count of transactions per month.
- Finally, output.saveAsTextFile("/user/ng311/Part_A_out") saves the output in the HDFS with file name Part_A_out.

Spark Job ID – http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1574975221160_8454/jobs/

Graph:

The output data from **Part_A_out** file is used to plot the below graph. The graph shows the trend of Total Transaction count per month.

- From Aug 2015 to Jan 2016 the transaction count is very low. From Feb 2016 to Feb 2017 there is a little increase in transactions.
- From Mar 2017 to Jan 2018 there is a major increase in the volume of transactions and Jan 2018 being the peak with 3,35,04,270 transactions.
- After Jan 2018 the number of transactions decreases and increases marginally.



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

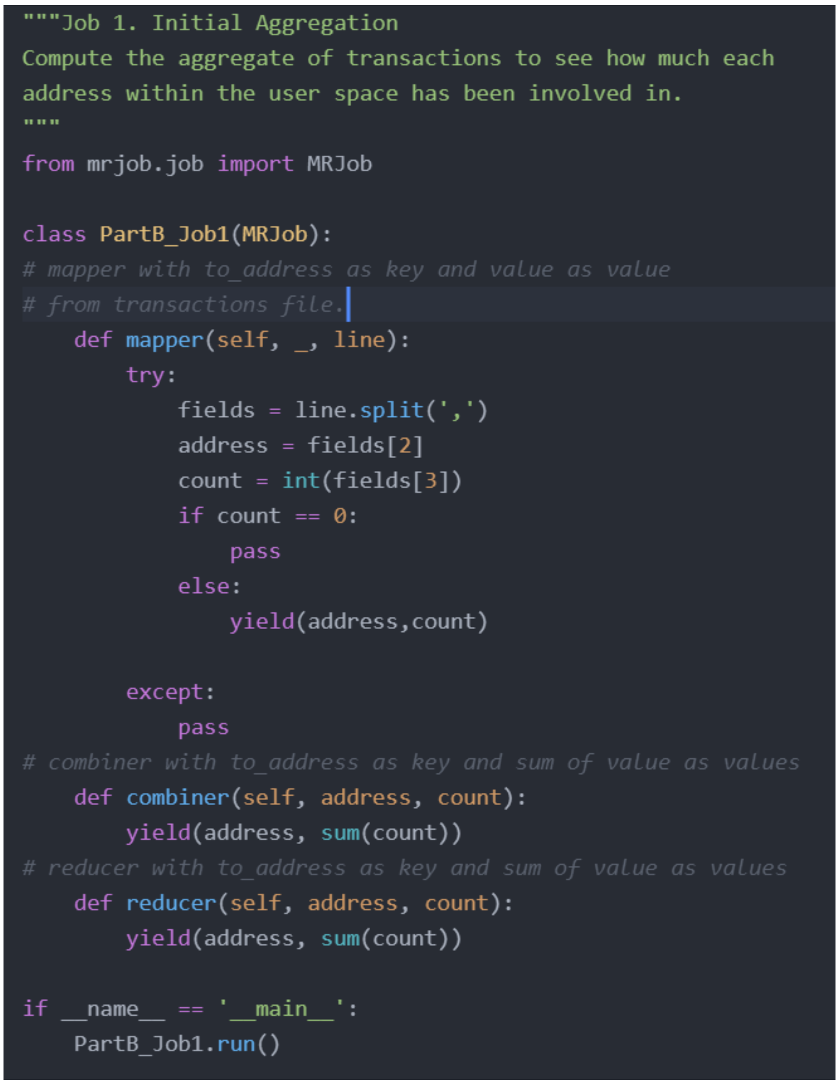
To workout which services are the most popular, you will first have to aggregate transactions to see how much each address within the user space has been involved in. You will want to aggregate value for addresses in the to_address field. This will be similar to the wordcount that we saw in Lab 1 and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

Approach:

As given, in order to calculate the aggregation of transactions the **value** field is aggregated for addresses in the **to_address** field from **transactions** file.

Code:

Hadoop Map/Reduce program to calculate the initial aggregation is given below. Program name is Part_B_Job1_Hadoop.py.



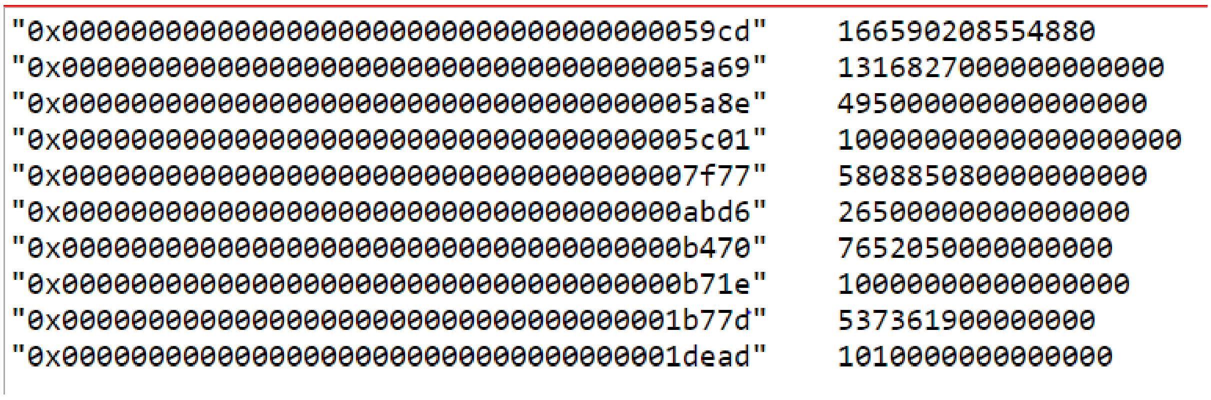
```python
"""Job 1. Initial Aggregation
Compute the aggregate of transactions to see how much each
address within the user space has been involved in.
"""
from mrjob.job import MRJob

class PartB_Job1(MRJob):
# mapper with to_address as key and value as value
# from transactions file.
    def mapper(self, _, line):
        try:
            fields = line.split(',')
            address = fields[2]
            count = int(fields[3])
            if count == 0:
                pass
            else:
                yield(address,count)

        except:
            pass
# combiner with to_address as key and sum of value as values
    def combiner(self, address, count):
        yield(address, sum(count))
# reducer with to_address as key and sum of value as values
    def reducer(self, address, count):
        yield(address, sum(count))

if __name__ == '__main__':
    PartB_Job1.run()
```

Code Explanation:

- Map 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.

Hadoop Job ID -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574935476688_0398/

Below are few lines of output from Job 1,



```
"0x00000000000000000000000000000000000059cd"    166590208554880
"0x00000000000000000000000000000000000005a69"    1316827000000000000
"0x00000000000000000000000000000000000005a8e"    495000000000000000
"0x00000000000000000000000000000000000005c01"    10000000000000000000
"0x00000000000000000000000000000000000007f77"    580885080000000000
"0x0000000000000000000000000000000000000abd6"    26500000000000000
"0x0000000000000000000000000000000000000b470"    7652050000000000
"0x0000000000000000000000000000000000000b71e"    10000000000000000
"0x000000000000000000000000000000000001b77d"    537361900000000
"0x000000000000000000000000000000000001dead"    1010000000000000
```

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

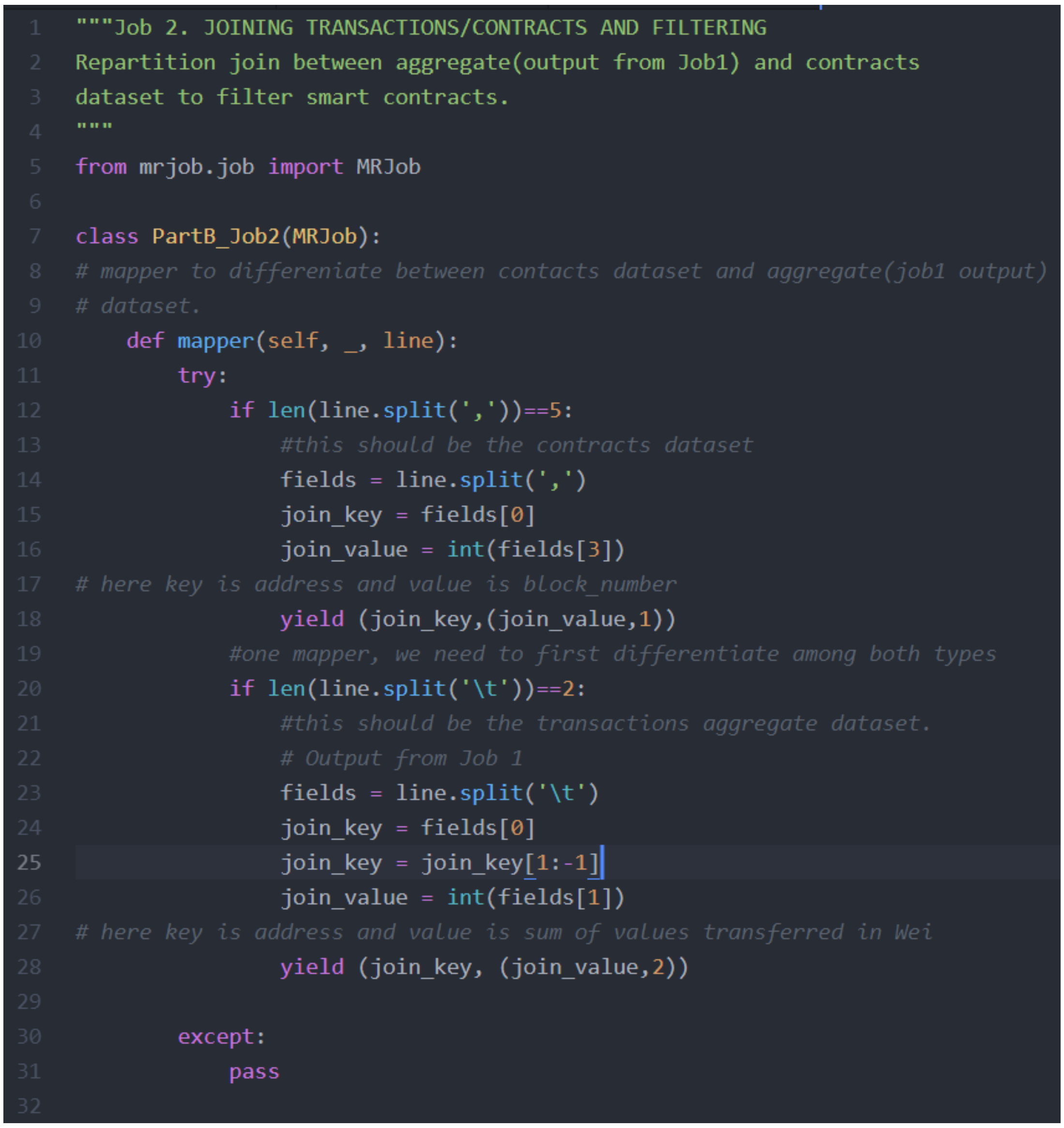
Secondly, in the reducer, if the address for a given aggregate from Job 1 was not present within contracts this should be filtered out as it is a user address and not a smart contract.

Approach:

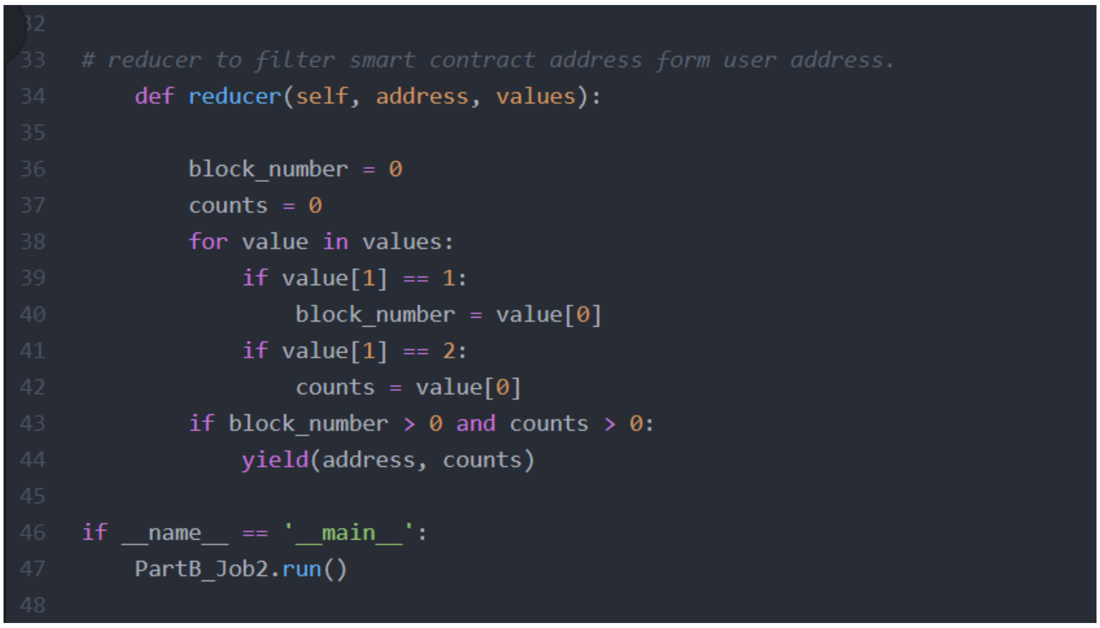
As given, perform repartition join between contracts dataset and transactions aggregate dataset (output from job 1) to filter out user address from smart contract address. The final list of records will have only address that belongs to smart contracts.

Code:

Program name is Part_B_Job2_Hadoop.py



```python
"""Job 2. JOINING TRANSACTIONS/CONTRACTS AND FILTERING
Repartition join between aggregate(output from Job1) and contracts
dataset to filter smart contracts.
"""
from mrjob.job import MRJob

class PartB_Job2(MRJob):
# mapper to differeniate between contacts dataset and aggregate(job1 output)
# dataset.
    def mapper(self, _, line):
        try:
            if len(line.split(','))==5:
                #this should be the contracts dataset
                fields = line.split(',')
                join_key = fields[0]
                join_value = int(fields[3])
# here key is address and value is block_number
                yield (join_key,(join_value,1))
            #one mapper, we need to first differentiate among both types
            if len(line.split('\t'))==2:
                #this should be the transactions aggregate dataset.
                # Output from Job 1
                fields = line.split('\t')
                join_key = fields[0]
                join_key = join_key[1:-1]
                join_value = int(fields[1])
# here key is address and value is sum of values transferred in Wei
                yield (join_key, (join_value,2))

        except:
            pass

```



```python

# reducer to filter smart contract address form user address.
    def reducer(self, address, values):

        block_number = 0
        counts = 0
        for value in values:
            if value[1] == 1:
                block_number = value[0]
            if value[1] == 2:
                counts = value[0]
        if block_number > 0 and counts > 0:
            yield(address, counts)

if __name__ == '__main__':
    PartB_Job2.run()

```

Code Explanation:

- Map reduce job to perform repartition join between contracts dataset and transaction aggregate dataset (output from job 1).
- In the mapper we are differentiating the two input files by checking the number of fields present in the file. If the number of fields is 5 its contracts dataset and of number of fields is 2 its transactions aggregate dataset (output from job 1).
- The first if condition in mapper recognises the contracts dataset where we specify **key as address**(fields[0]) and **value as block_number and '1'.** 1 is hardcoded to identify that the value is from contracts dataset at the reducer.
- The second if condition in mapper recognises the transactions aggregate dataset where we specify **key as address** and **value as aggregate count (sum of values in Wei).**
- The mapper takes records only if both the keys matches. That is only if address from both the dataset matches. By this way we filter smart contract address.
- In the reducer using for loop I identified the **aggregate values** count from the set of values yielded by mapper. If value[1] == 2 then **counts** is the **aggregate value.**
- Finally, the **key** which is the **smart contract address** and **value** which is the **aggregate value** is yielded in the reducer which gives us the aggregate values of all smart contracts.

Hadoop Job ID -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574935476688_0440/

Below are few lines of output from Job 2,

```
"0x0009203c16bd97898d526fe6ba1a56a002faed81"    470000000000000000000
"0x00094cdbff7bb5e60dde61d8400700cee968dc08"    110438750000000000
"0x00094de162491ce3555f78d41fd1d9c4a2dc6155"    316430140000000000
"0x00095c42ef1d1f5d2be4ee7fe277a4b0cc5408d9"    115441000000000000
"0x0009ad67250df405dfbe17ed20a435f01752b7eb"    2132991300000000000
"0x0009c3a88872d923d5624b1a1f0a1bc4f5705b3c"    10000000000000000
"0x0009d531b8560d8292c820e61600a8853a412236"    3288491779835122520
"0x0009ed6f934b570c4057d674b1f98b00f9eac5cc"    5449644244260000000
"0x0009fc17eb5455b89da09915d306c1d35055a85f"    400000000000000000
"0x000a03ba7648bb2e321201ec4a4bb2c9f05d02cb"    4210255310000000000
"0x000a1eb88e128c82cd1cdccd11ec7c417785e1b4"    765369900000000000
```

JOB 3 - TOP TEN

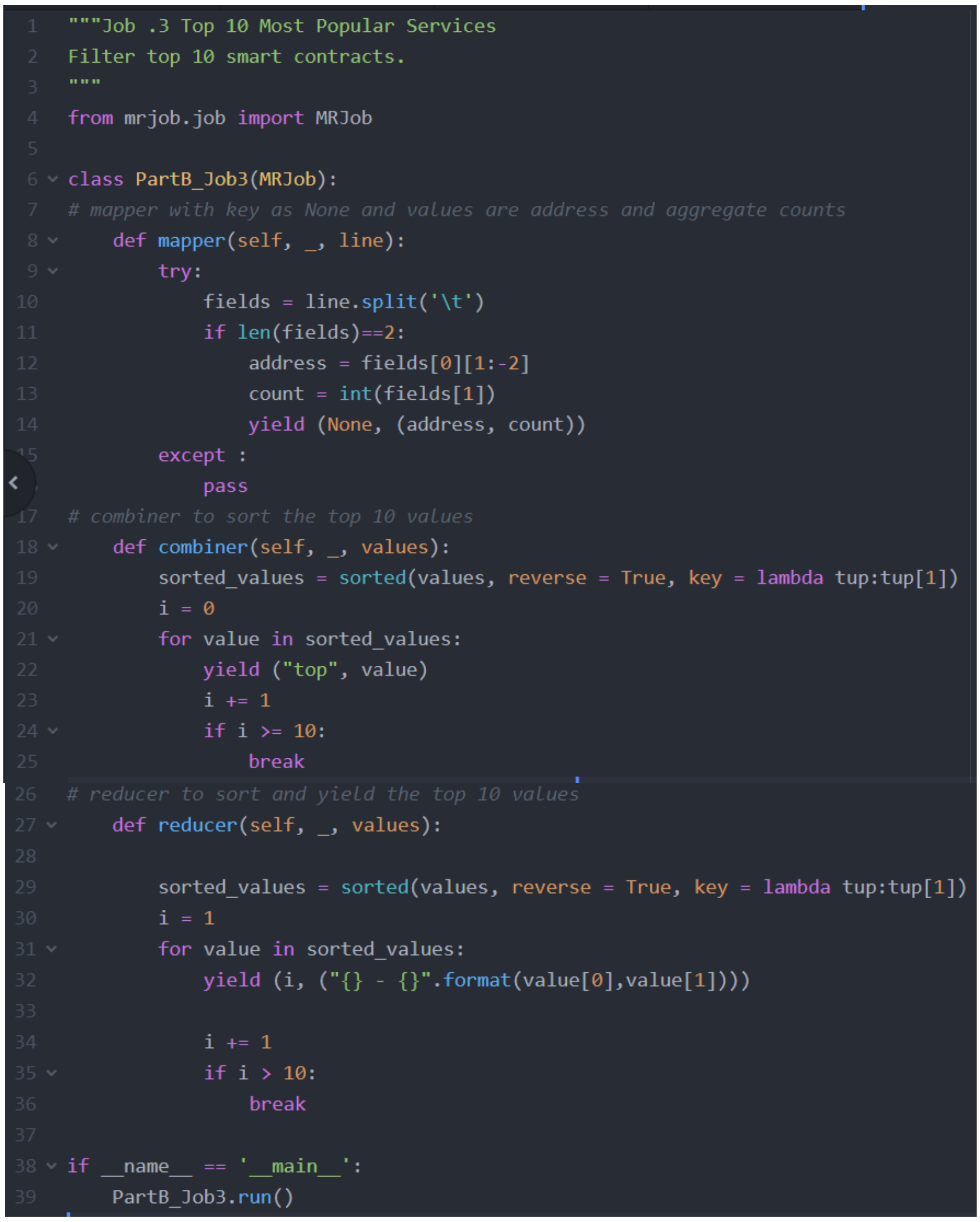
Finally, the third job will take as input the now filtered address aggregates and sort these via a top ten reducer, utilising what you have learned from lab 4.

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.

Code:

Program name is Part_B_Job3_Hadoop.py



```python
"""Job .3 Top 10 Most Popular Services
Filter top 10 smart contracts.
"""
from mrjob.job import MRJob

class PartB_Job3(MRJob):
# mapper with key as None and values are address and aggregate counts
    def mapper(self, _, line):
        try:
            fields = line.split('\t')
            if len(fields)==2:
                address = fields[0][1:-2]
                count = int(fields[1])
                yield (None, (address, count))
        except :
            pass
# combiner to sort the top 10 values
    def combiner(self, _, values):
        sorted_values = sorted(values, reverse = True, key = lambda tup:tup[1])
        i = 0
        for value in sorted_values:
            yield ("top", value)
            i += 1
            if i >= 10:
                break
# reducer to sort and yield the top 10 values
    def reducer(self, _, values):

        sorted_values = sorted(values, reverse = True, key = lambda tup:tup[1])
        i = 1
        for value in sorted_values:
            yield (i, ("{} - {}".format(value[0],value[1])))

            i += 1
            if i > 10:
                break

if __name__ == '__main__':
    PartB_Job3.run()
```

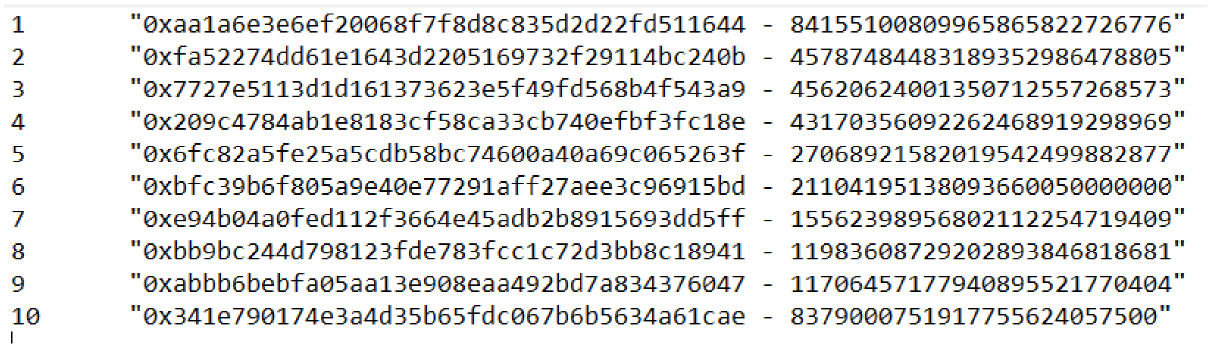
Code Explanation:

- Map reduce job to filter out top 10 most popular services obtained as a result from Job 2 output.
- In the mapper the lines are split by tab since the input file is tab separated. **Key is None** and value is the **address and sum of aggregate values.** Key is none because we need to sort the values based on the aggregate count.

- A combiner is used to speed up the sorting process. In combiner sort the values in ascending order by giving **reverse = True.** This will return the top values first. After this condition has been set a for loop is used to iterate between all the records and yield the sort the values in ascending order.
- Finally, in reducer the same combiner operation is performed where key value pairs from all the combiners are sorted once again to obtain the exact result. A for loop is used to iterate between all the records from combiner and the result is displayed in the top 10 order. In order to display only top 10 values the for loop is terminated if the iteration count reaches 11.
- Output is the top 10 most popular services.

Hadoop Job ID -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_0647/

Below are top 10 services (output from Job3),



```
1       "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd511644 - 84155100809965865822726776"
2       "0xfa52274dd61e1643d2205169732f29114bc240b - 45787484483189352986478805"
3       "0x7727e5113d1d161373623e5f49fd568b4f543a9 - 45620624001350712557268573"
4       "0x209c4784ab1e8183cf58ca33cb740efbf3fc18e - 43170356092262468919298969"
5       "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f - 27068921582019542499882877"
6       "0xbfc39b6f805a9e40e77291aff27aee3c96915bd - 21104195138093660050000000"
7       "0xe94b04a0fed112f3664e45adb2b8915693dd5ff - 15562398956802112254719409"
8       "0xbb9bc244d798123fde783fcc1c72d3bb8c18941 - 11983608729202893846818681"
9       "0xabbb6bebfa05aa13e908eaa492bd7a834376047 - 11706457177940895521770404"
10      "0x341e790174e3a4d35b65fdc067b6b5634a61cae - 8379000751917755624057500"
```

PART C. DATA EXPLORATION (30%)

MISCELLANEOUS ANALYSIS

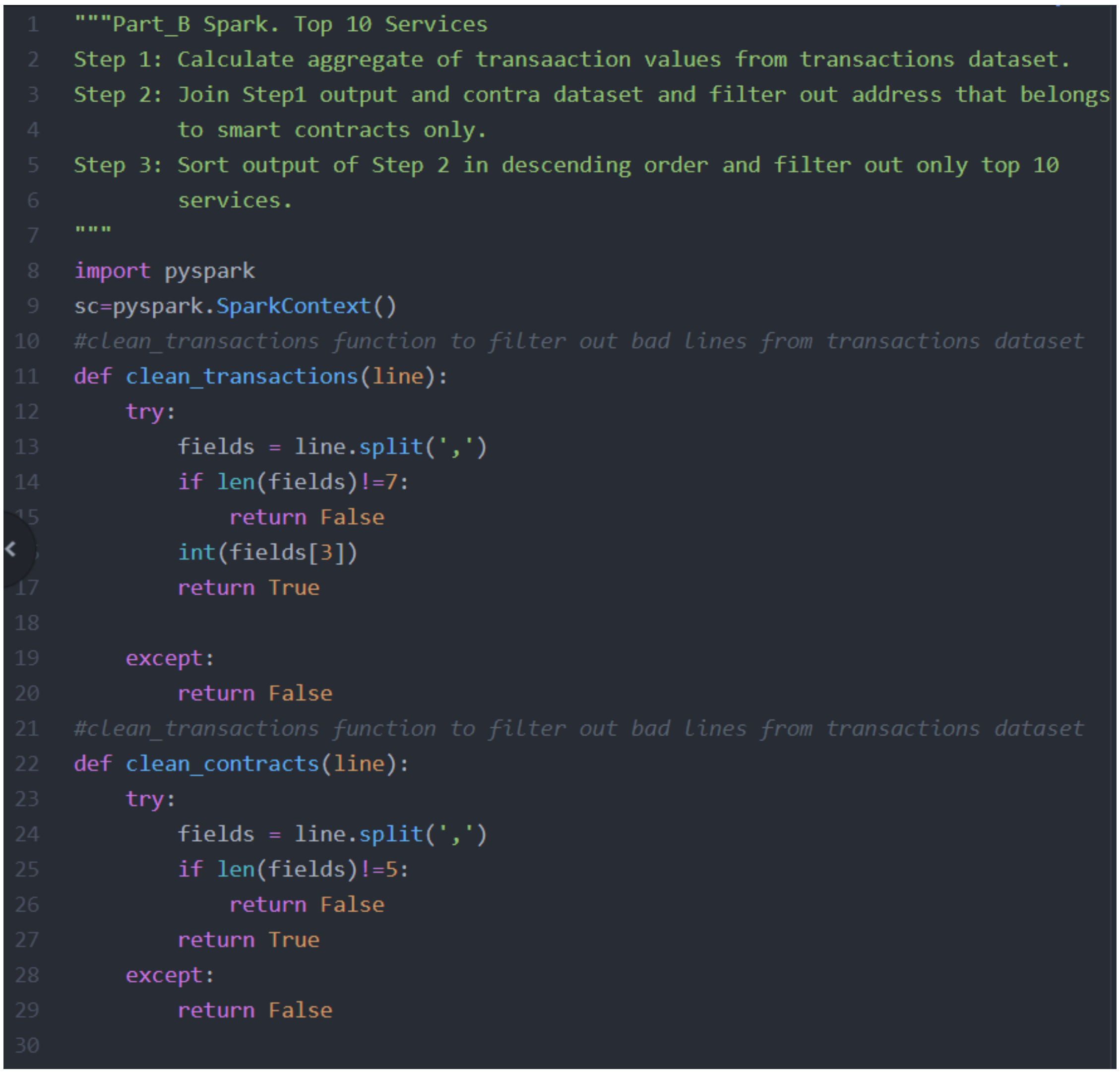
Comparative Evaluation Reimplement Part B in Spark (if your original was MRJob, or vice versa). How does it run in comparison? Keep in mind that to get representative results you will have to run the job multiple times, and report median/average results. Can you explain the reason for these results?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

Approach:

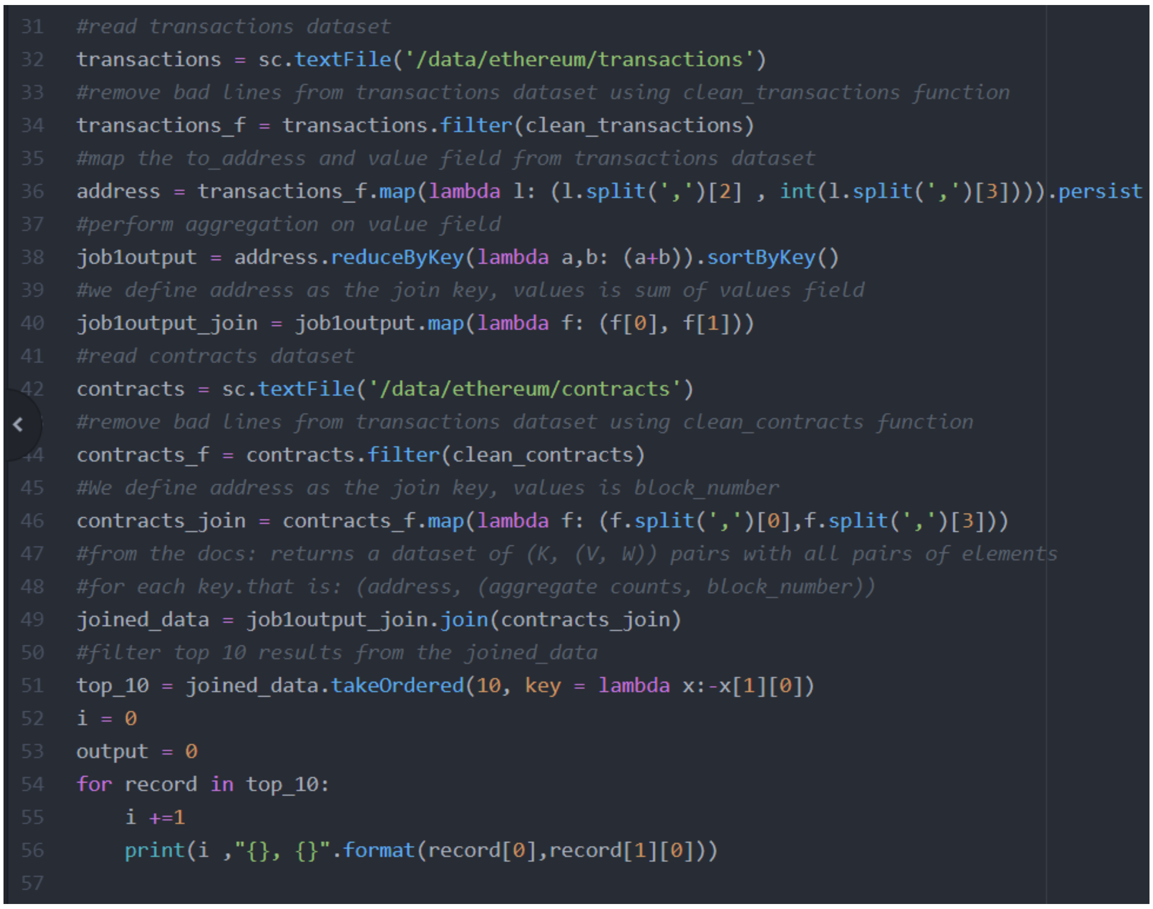
I have decided to implement Part B with a single spark Job. The operations of all the 3 jobs will be performed by a single spark job. First step is to calculate the aggregate counts for all transactions in the **TRANSACTIONS dataset**. Second step is to filter out the smart contracts address from user address. Third step is to sort the address and filter out top 10 services.

Code:

Program name is Part_B_Spark.py



```python
"""Part_B Spark. Top 10 Services
Step 1: Calculate aggregate of transaaction values from transactions dataset.
Step 2: Join Step1 output and contra dataset and filter out address that belongs
        to smart contracts only.
Step 3: Sort output of Step 2 in descending order and filter out only top 10
        services.
"""
import pyspark
sc=pyspark.SparkContext()
#clean_transactions function to filter out bad lines from transactions dataset
def clean_transactions(line):
    try:
        fields = line.split(',')
        if len(fields)!=7:
            return False
        int(fields[3])
        return True

    except:
        return False
#clean_transactions function to filter out bad lines from transactions dataset
def clean_contracts(line):
    try:
        fields = line.split(',')
        if len(fields)!=5:
            return False
        return True
    except:
        return False

```



```python
#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#remove bad lines from transactions dataset using clean_transactions function
transactions_f = transactions.filter(clean_transactions)
#map the to_address and value field from transactions dataset
address = transactions_f.map(lambda l: (l.split(',')[2] , int(l.split(',')[3]))).persist
#perform aggregation on value field
job1output = address.reduceByKey(lambda a,b: (a+b)).sortByKey()
#we define address as the join key, values is sum of values field
job1output_join = job1output.map(lambda f: (f[0], f[1]))
#read contracts dataset
contracts = sc.textFile('/data/ethereum/contracts')
#remove bad lines from transactions dataset using clean_contracts function
contracts_f = contracts.filter(clean_contracts)
#We define address as the join key, values is block_number
contracts_join = contracts_f.map(lambda f: (f.split(',')[0],f.split(',')[3]))
#from the docs: returns a dataset of (K, (V, W)) pairs with all pairs of elements
#for each key.that is: (address, (aggregate counts, block_number))
joined_data = job1output_join.join(contracts_join)
#filter top 10 results from the joined_data
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:

- First line reads the transaction dataset using pyspark's sparkcontext function. Second line filters any bad lines from transactions dataset using the user defined function clean_transactions.
- In third line using spark's lambda function we map the key as **address** and value as **values** from transaction dataset. In fourth line we use spark's **reduceByKey** function to calculate the aggregate of transaction **values.** This output is kept in memory .
- In fifth line we map the key as address and value as **aggregate values** from the output of previous operation. Spark's in-memory processing is made use here. These values are stored in variable **job1output_join.**
- Sixth line reads the contract dataset. Seventh line filters out the bad lines from contracts dataset.
- Eighth line maps the key as **address** and value as **block_number** from contracts dataset using spark's **lambda** function.
- Ninth line performs the join operation using spark's **join** operation. Variable **joined_data** is the result of the joined dataset.
- Tenth line performs the sorting operation and filtering only 10 values using spark's **takeOrdered** function. The symbol **'-'** in lambda function **x:-x[1][0]** sorts the values in descending order leaving top value at first.
- The final step is to format the values after sorting and print the top 10 services.

Spark ID – http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1301/jobs/

Below are the top 10 services obtained from Spark Job,

```
(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: 8379000751917755624057500')
```

Performance Evaluation:

Spark Job ID's –

http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1301/jobs/

http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1313/jobs/

http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1323/jobs/

http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1329/jobs/

http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_1334/jobs/

Hadoop Job ID's –

Job 1 - http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1339/

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1411/

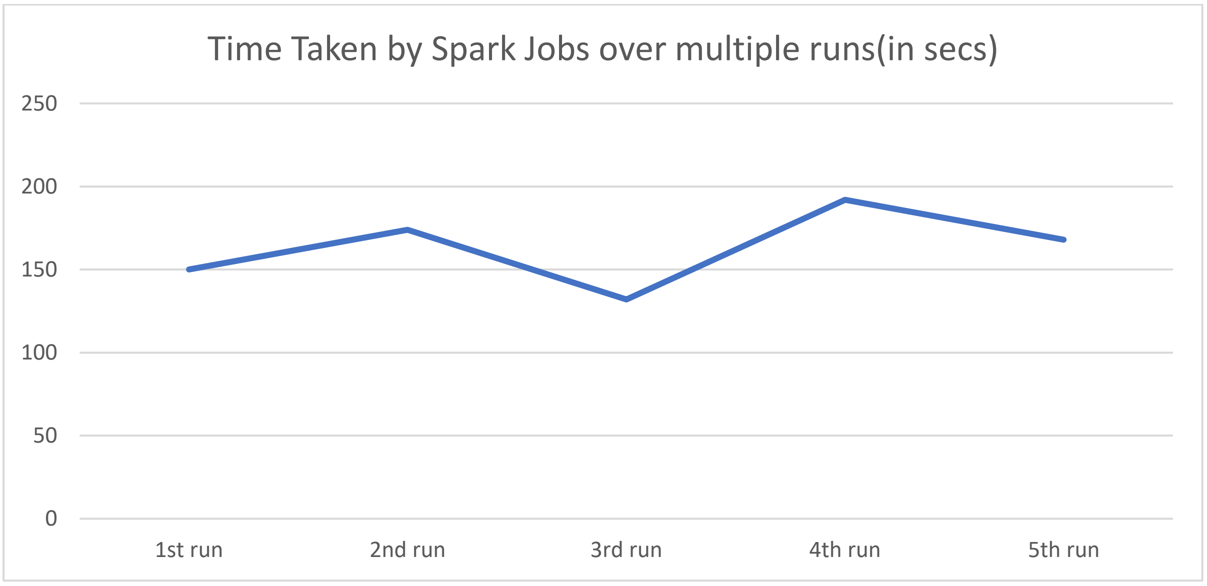
Job 2 - http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1387/

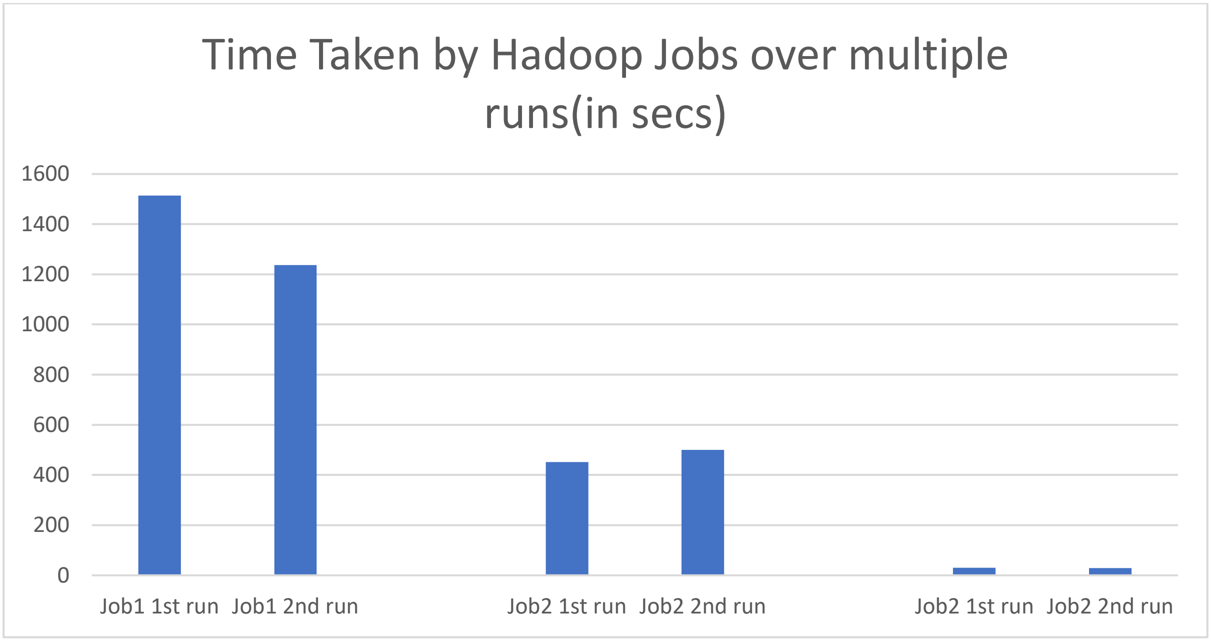
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1562/

Job 3 –

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1404/

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1686/





- Comparing the results of Hadoop and Spark jobs, spark jobs seems to perform faster for this task.
- This is due to Spark's in-memory processing where the intermediate results can be stored in RDD and the next transformation or action can be applied to that RDD.
- Every time we execute Hadoop jobs the output data must be written and read to HDFS but in Spark we can directly read it from RDD.
- All three jobs of Part B can be implemented in a single Spark Job.
- Multiple map and reduce steps in a single Hadoop job can be implemented for this task but the time taken to process the records by map reduce job will be more. The

amount of shuffle and sort that takes place during join operation is large which reduces the job’s performance.

- 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.

MISCELLANEOUS ANALYSIS

Gas Guzzlers: 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. (15/30).

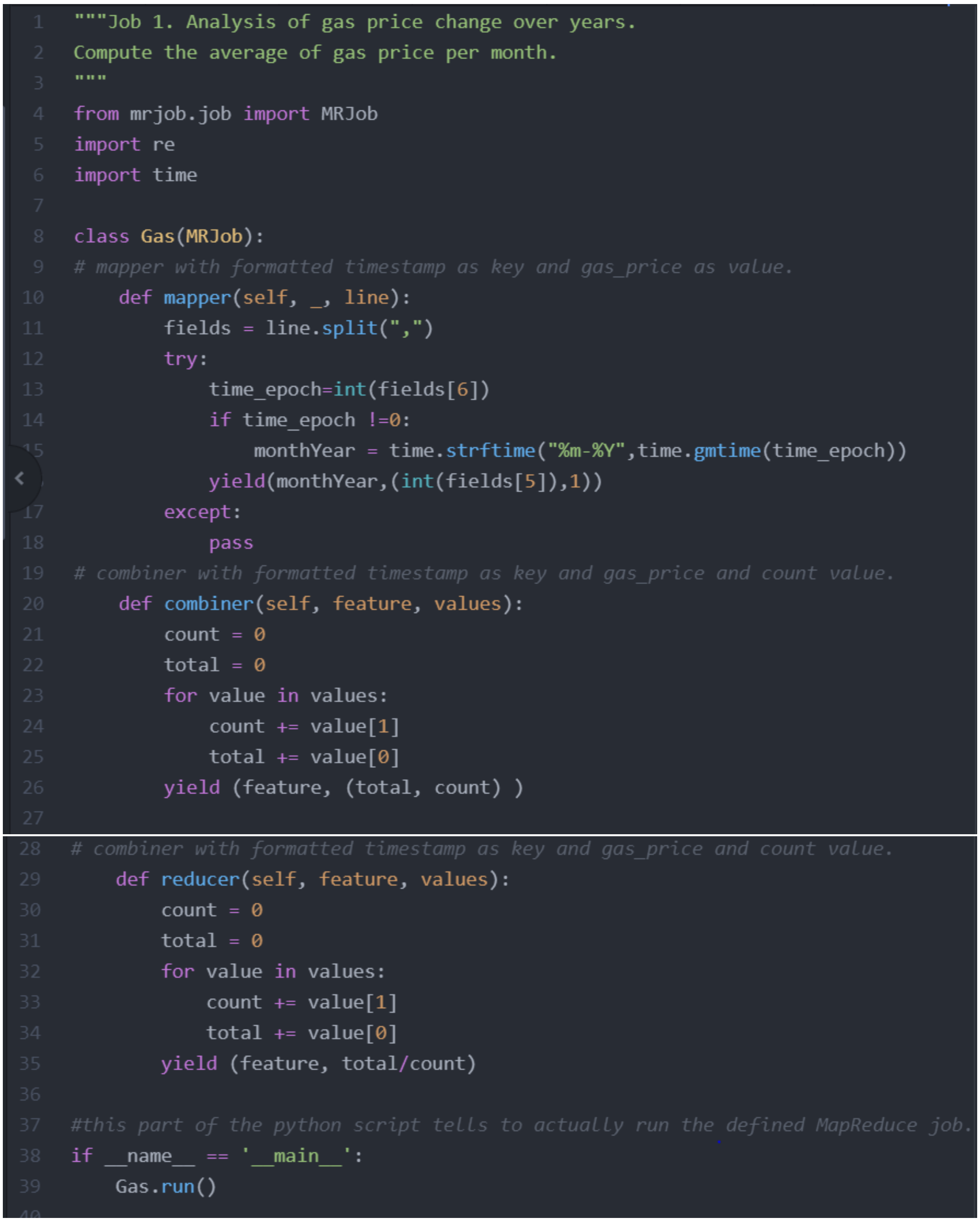
Part 1: Time series analysis of gas price change over the years(in Wei).

Approach:

In order to calculate the gas price change over the years, I have used the fields **gas_price** and **block_timestamp** values from the transaction dataset. The average value of gas price per month gives the time series analysis of how the price has changed over the years.

Code:

The program is gas_part1.


```python
"""Job 1. Analysis of gas price change over years.
Compute the average of gas price per month.
"""
from mrjob.job import MRJob
import re
import time

class Gas(MRJob):
# mapper with formatted timestamp as key and gas_price as value.
    def mapper(self, _, line):
        fields = line.split(",")
        try:
            time_epoch=int(fields[6])
            if time_epoch !=0:
                monthYear = time.strftime("%m-%Y",time.gmtime(time_epoch))
            yield(monthYear,(int(fields[5]),1))
        except:
            pass
# combiner with formatted timestamp as key and gas_price and count value.
    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total, count) )

# combiner with formatted timestamp as key and gas_price and count value.
    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, total/count)

#this part of the python script tells to actually run the defined MapReduce job.
if __name__ == '__main__':
    Gas.run()
```

Hadoop Job ID –
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1575381276332_1508/

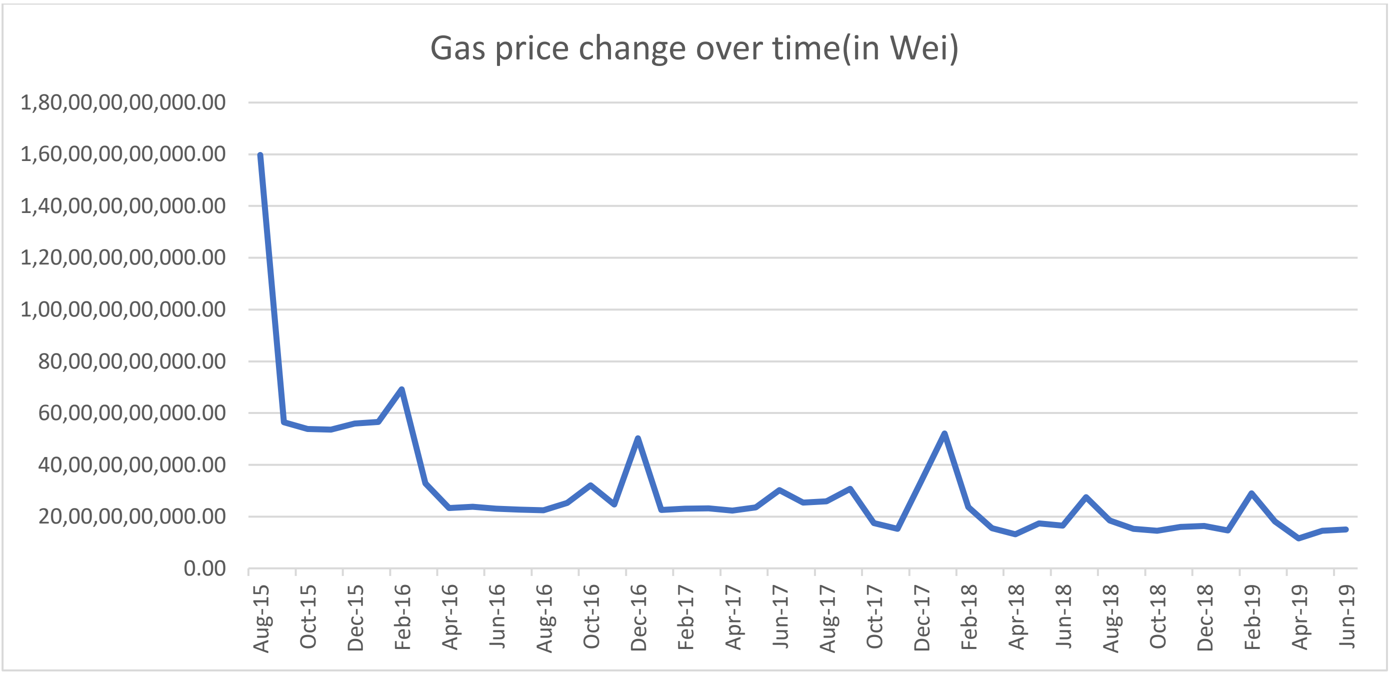
Code Explanation:

- To analyse 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:

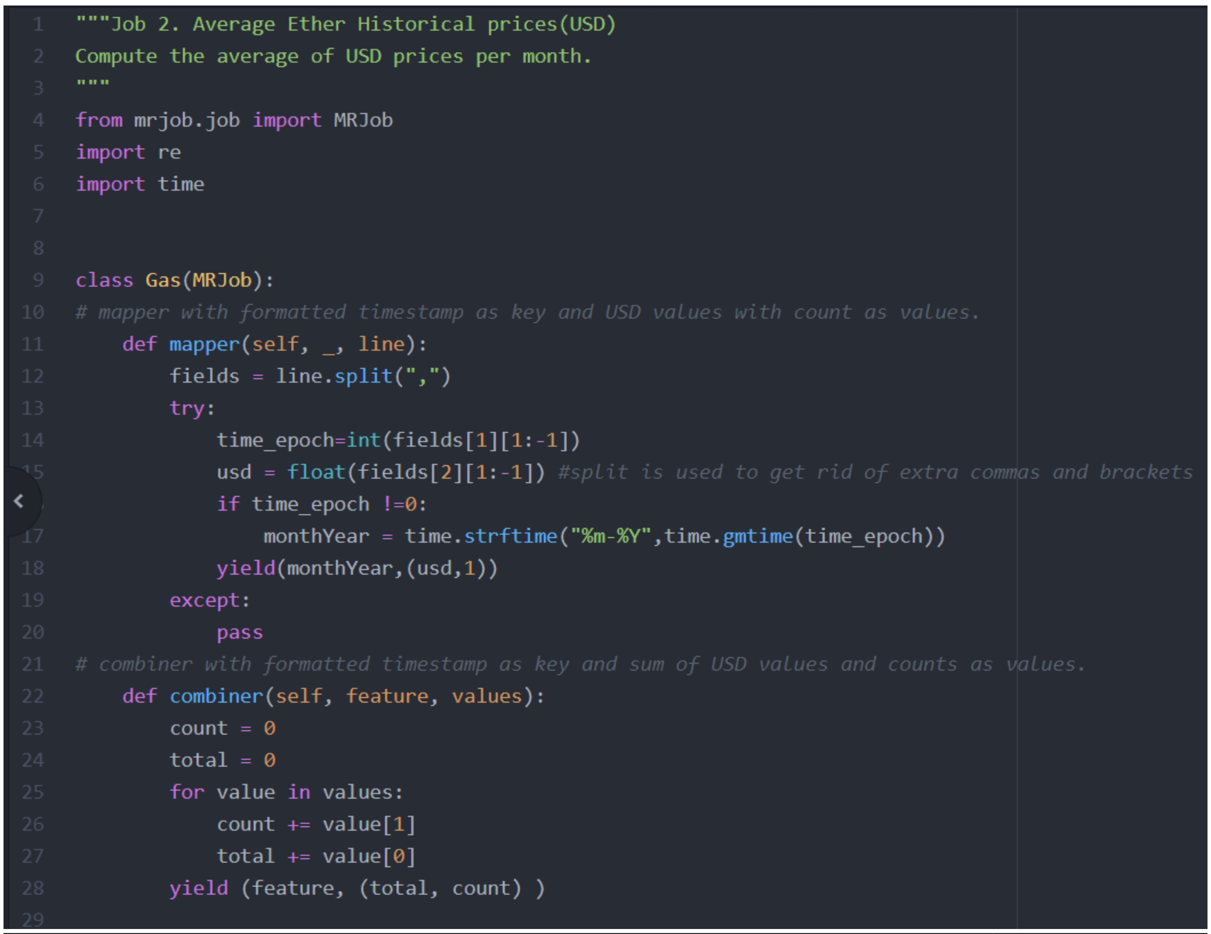
The obtained results from the map reduce job is plotted using excel and below is the time series analysis graph of gas price change over years,



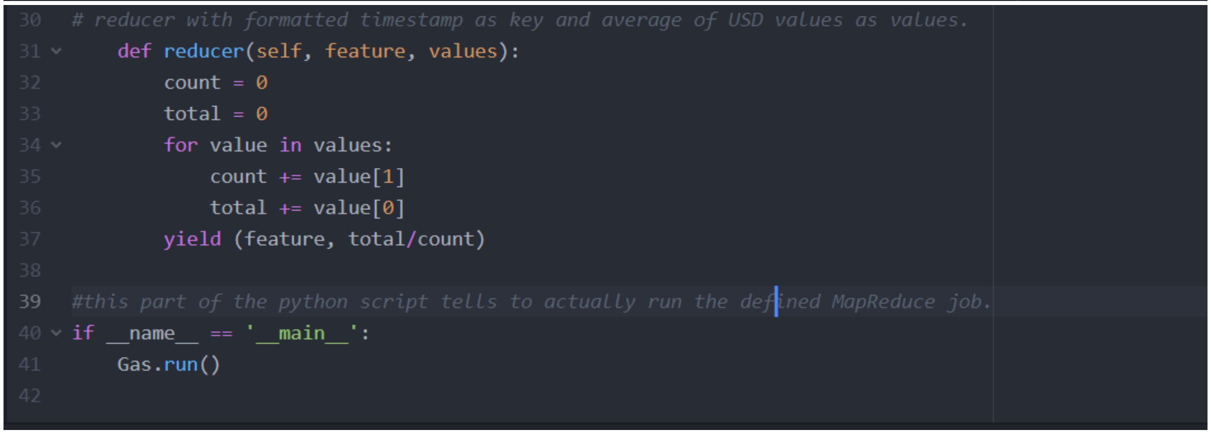
- The plot shows that during the inception of ethereum in Aug 2015 the price has been more and next month there is a sudden decrease in the value.
- After September 2015 the gas price value seems to be steady with minor variations until Feb 2016.
- After Feb 2016 the gas price decreases gradually and almost seems to be steady until Jun 2019 with small fluctuations.

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

- Using the previous results, I converted the Wei values to USD values.
- First, I converted the Wei values to ether. 1 **ether** = 1000000000000000000 **wei**. To convert the Wei values to USD values I need the exchange rate historical data for Ether to USD.
- The exchange rate historical data was gathered from https://etherscan.io/chart/etherprice?output=csv .
- The exchange rate values are stock rates per day and the rates vary with a single day until stock closes. Since my previous calculations in Wei are the average results per month, I need the average of USD prices per month to do the conversion.
- I used a map reduce job to convert the daily USD rates to average USD rates per month. Below is the map reduce job used. Program name is gasusd.py



```python
"""Job 2. Average Ether Historical prices(USD)
Compute the average of USD prices per month.
"""
from mrjob.job import MRJob
import re
import time


class Gas(MRJob):
# mapper with formatted timestamp as key and USD values with count as values.
    def mapper(self, _, line):
        fields = line.split(",")
        try:
            time_epoch=int(fields[1][1:-1])
            usd = float(fields[2][1:-1]) #split is used to get rid of extra commas and brackets
            if time_epoch !=0:
                monthYear = time.strftime("%m-%Y",time.gmtime(time_epoch))
            yield(monthYear,(usd,1))
        except:
            pass
# combiner with formatted timestamp as key and sum of USD values and counts as values.
    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total, count) )

```

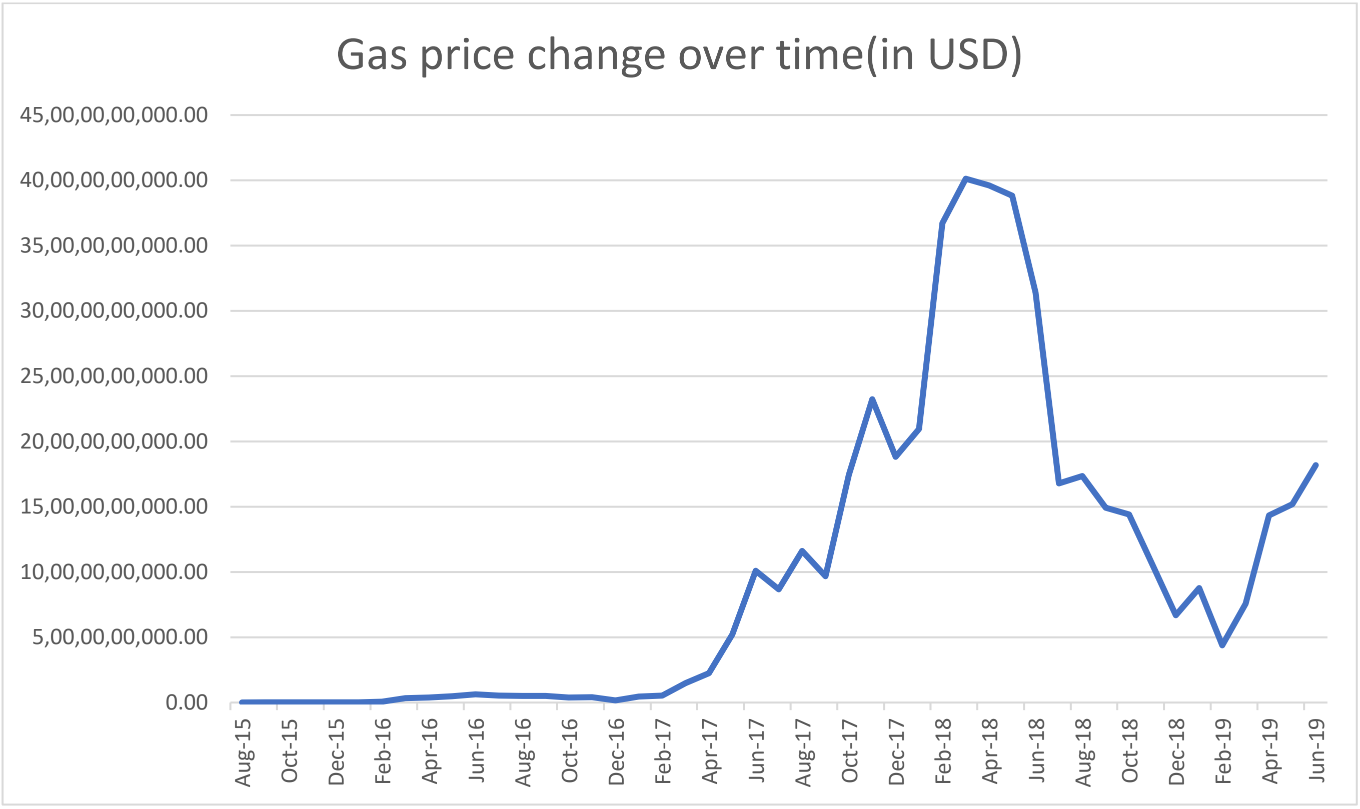


```python
# reducer with formatted timestamp as key and average of USD values as values.
    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, total/count)

#this part of the python script tells to actually run the defined MapReduce job.
if __name__ == '__main__':
    Gas.run()

```

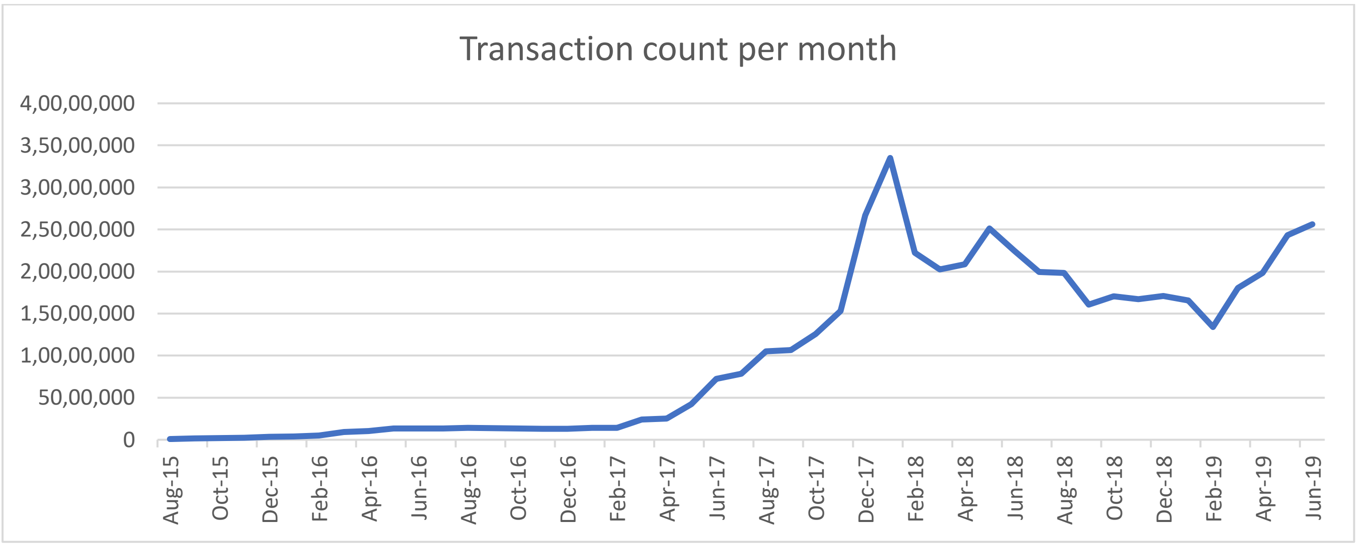
- Using the result of the map reduce job the average gas price per month in USD is calculated by multiplying the ether price with the map reduce output.

Hadoop Job ID -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_7284/

Gas price change over time in USD,



One interesting insight is that the above graph correlates with the transaction count per month graph. The USD prices seems to increase when there are a greater number of transactions. The trend seems to be similar. Once again below is the transaction count per month for reference.



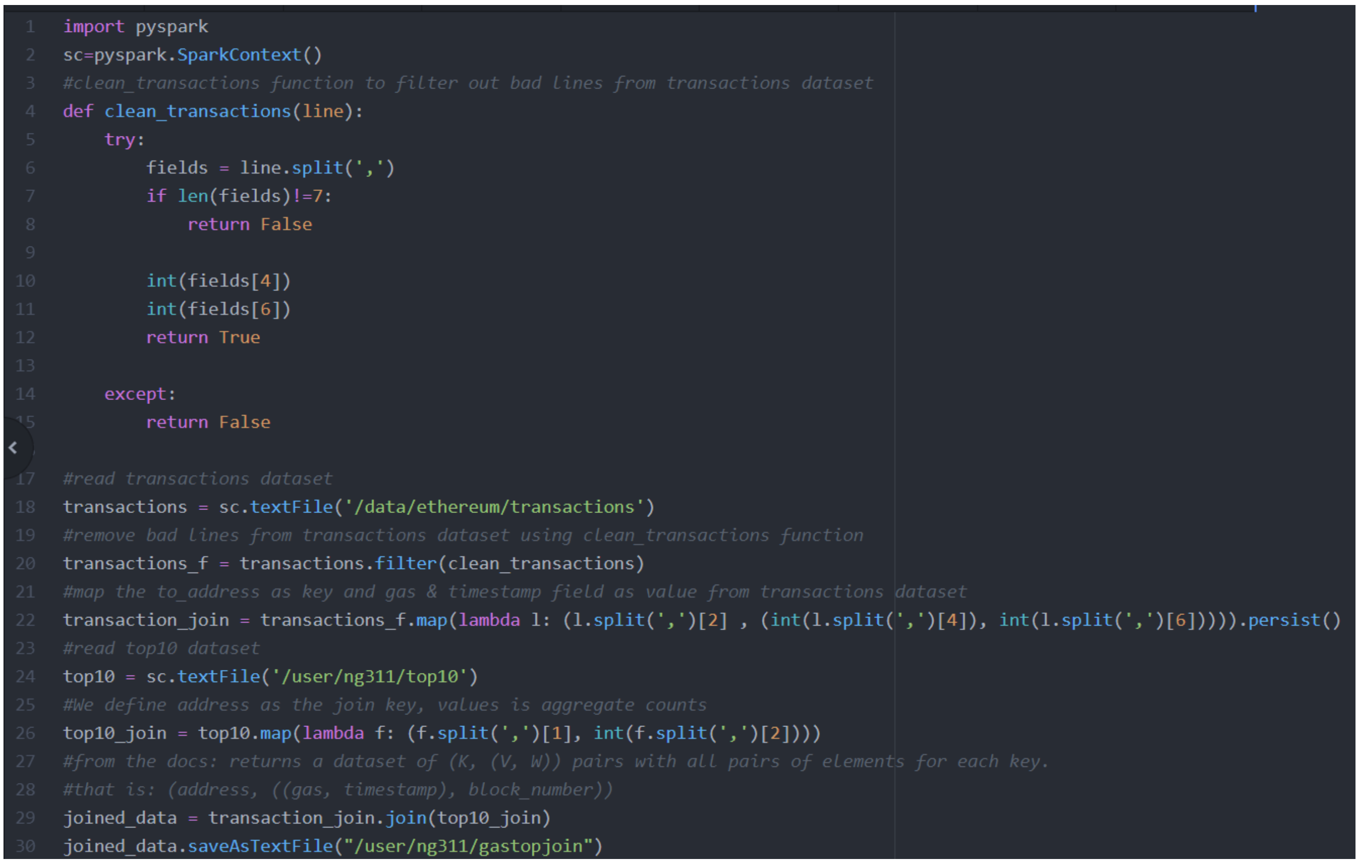
Part 2: Time series analysis of Gas consumed by top 10 smart contracts over the years.

Approach:

In order to calculate the amount of gas consumed by top 10 contracts over the years a replication join is performed between the top 10 smart contracts and transactions dataset. The output of replication join is top 10 addresses, its gas provided by sender and Unix timestamp values. After join operation the average of gas consumed over the years per address is calculated.

Step1: Perform join operation on Transactions dataset and Top 10 smart contracts dataset.

Join Spark Job: Program name is gastop.py



```
import pyspark
sc=pyspark.SparkContext()
#clean_transactions function to filter out bad lines from transactions dataset
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

#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#remove bad lines from transactions dataset using clean_transactions function
transactions_f = transactions.filter(clean_transactions)
#map the to_address as key and gas & timestamp field as value from transactions dataset
transaction_join = transactions_f.map(lambda l: (l.split(',')[2] , (int(l.split(',')[4]), int(l.split(',')[6])))).persist()
#read top10 dataset
top10 = sc.textFile('/user/ng311/top10')
#We define address as the join key, values is aggregate counts
top10_join = top10.map(lambda f: (f.split(',')[1], int(f.split(',')[2])))
#from the docs: returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key.
#that is: (address, ((gas, timestamp), block_number))
joined_data = transaction_join.join(top10_join)
joined_data.saveAsTextFile("/user/ng311/gastopjoin")
```

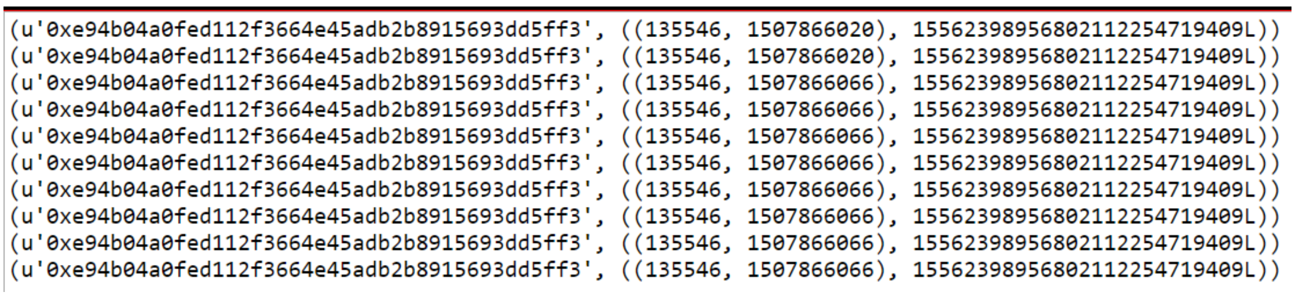
Spark Join Job ID –
http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1574975221160_7940/jobs/

Code Explanation:

- Spark job to perform join operation on top 10 smart contracts dataset and transactions dataset.
- Transactions dataset is read and map operation is performed. Key is to_address and values are gas and timestamp field. The result of map is stored in transactions_join variable.
- Top 10 smart contracts dataset is read and map operation is performed. Key is address and value is aggregate counts. The result of map is stored in top10_join variable.

- The data is joined using **join** operation and the key is address and values are gas, timestamp and aggregate value.
- The result of join operation is stored in the HDFS named **gastopjoin.**

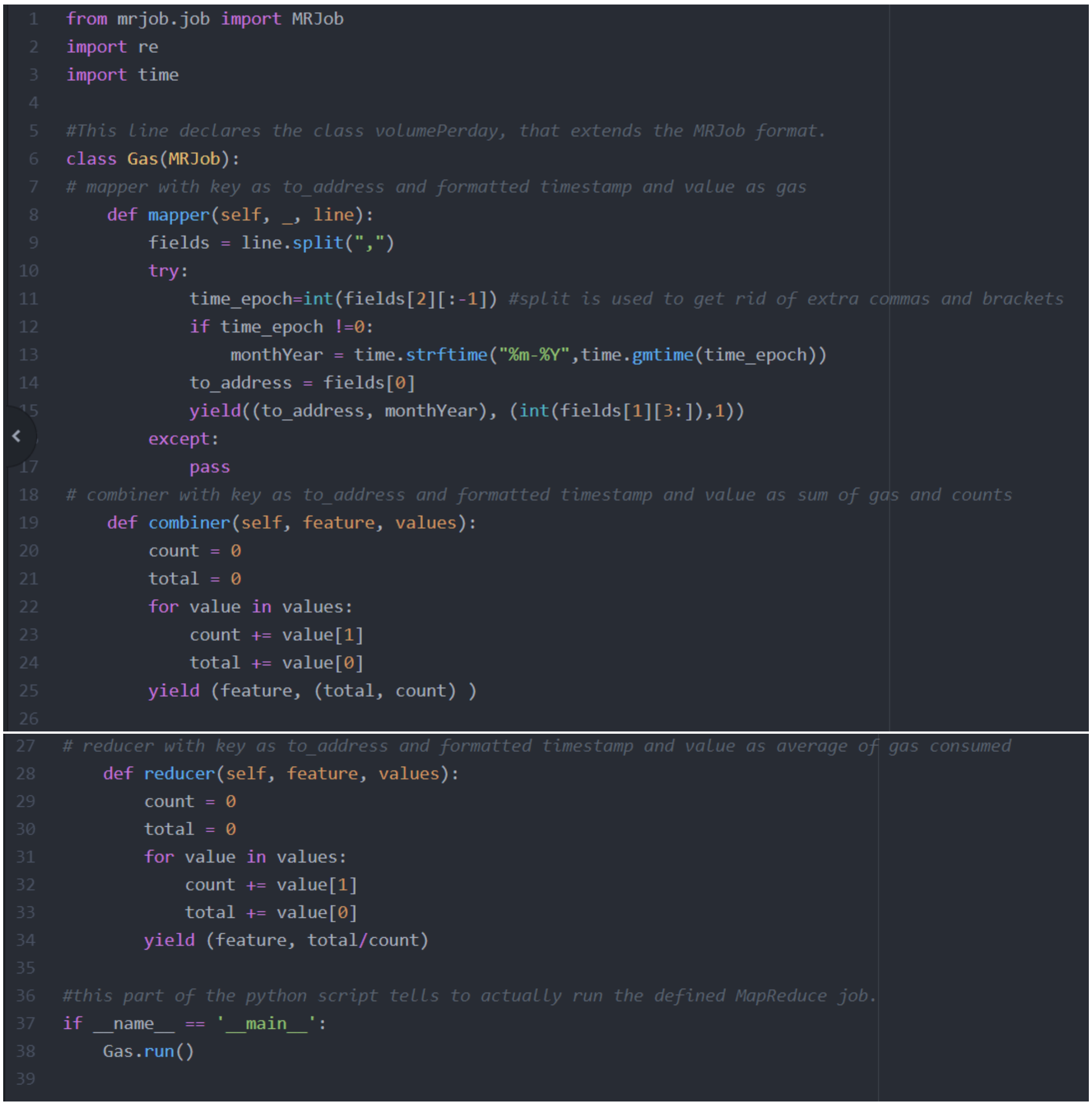
Few records of the join dataset are given below,



```
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866020), 155623989568021122547194 09L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866020), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', ((135546, 1507866066), 15562398956802112254719409L))
```

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

Average Calculation Hadoop Code: Program name is gas_hdp_top_avg.py



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

#This line declares the class volumePerday, that extends the MRJob format.
class Gas(MRJob):
# mapper with key as to_address and formatted timestamp and value as gas
    def mapper(self, _, line):
        fields = line.split(",")
        try:
            time_epoch=int(fields[2][:-1]) #split is used to get rid of extra commas and brackets
            if time_epoch !=0:
                monthYear = time.strftime("%m-%Y",time.gmtime(time_epoch))
            to_address = fields[0]
            yield((to_address, monthYear), (int(fields[1][3:]),1))
        except:
            pass
# combiner with key as to_address and formatted timestamp and value as sum of gas and counts
    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, (total, count) )

# reducer with key as to_address and formatted timestamp and value as average of gas consumed
    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield (feature, total/count)

#this part of the python script tells to actually run the defined MapReduce job.
if __name__ == '__main__':
    Gas.run()
```

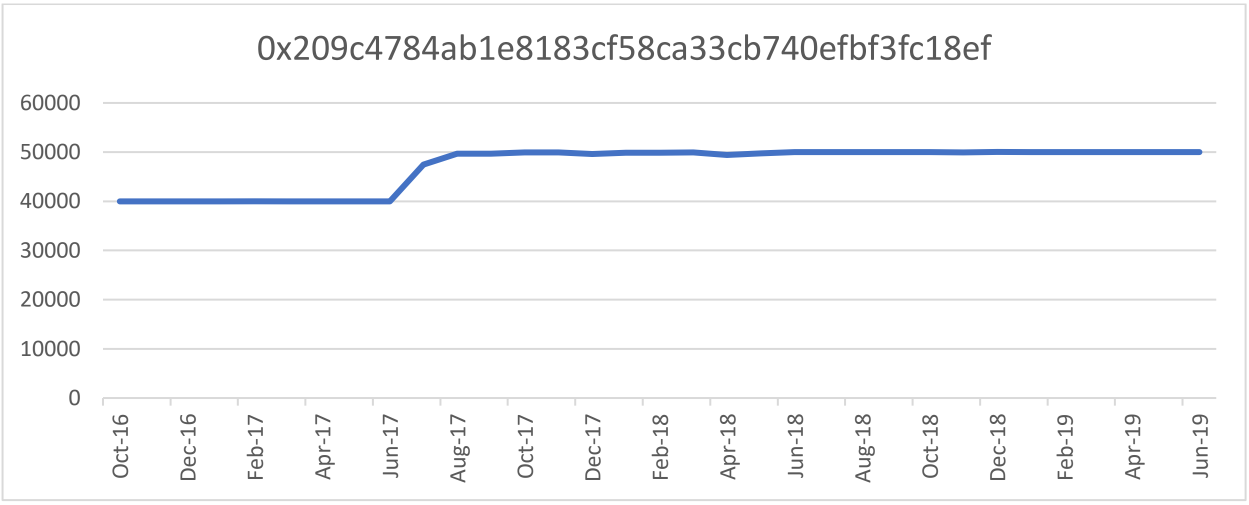
Code Explanation:

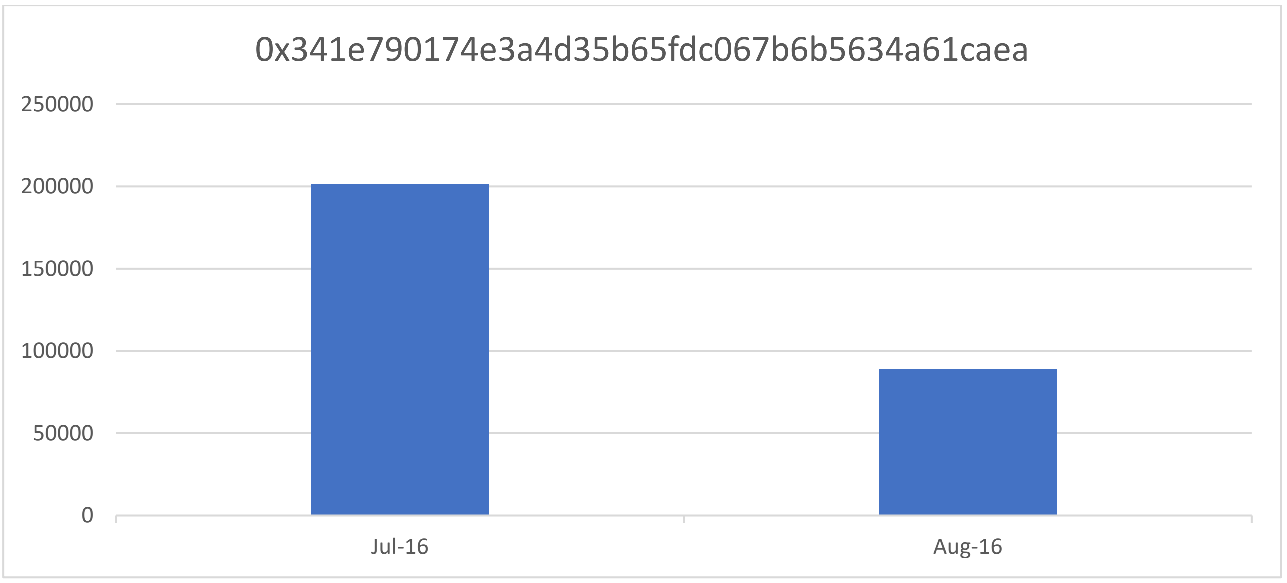
- Map reduce job to calculate the average of gas consumed per smart contract from its start time to end time.
- Mapper is defined with key as **to_address** and formatted timestamp. The Unix timestamp is converted to Gregorian format using **time** library. The values of the mapper are gas provided by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

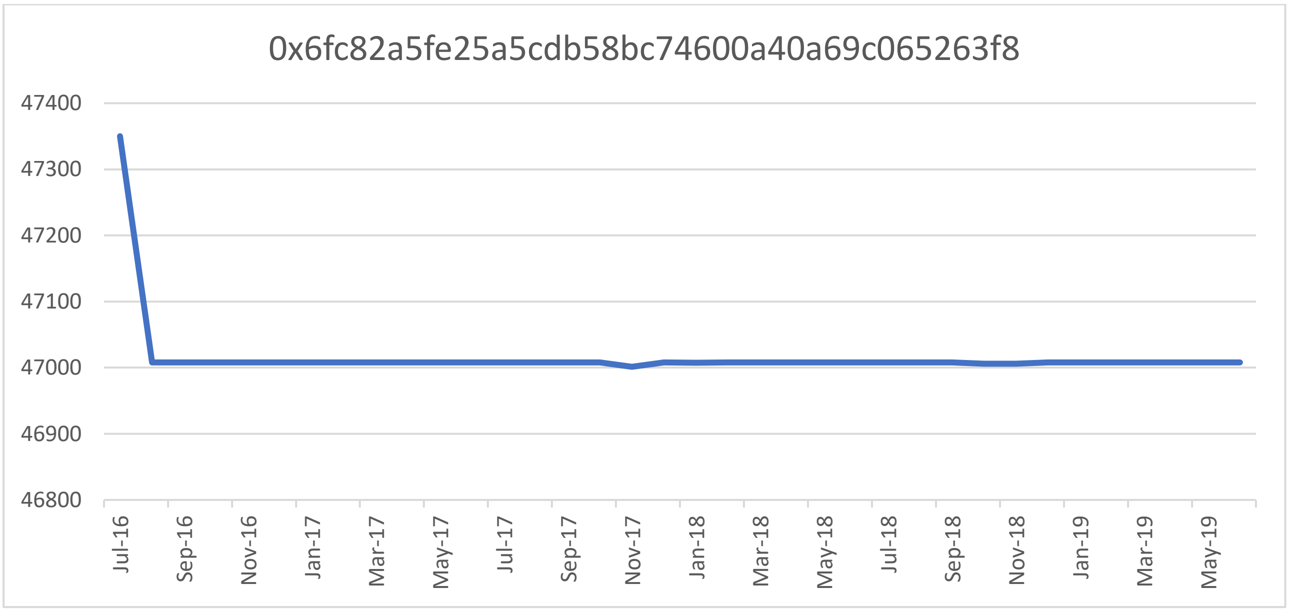
Hadoop Job ID -
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_8021/

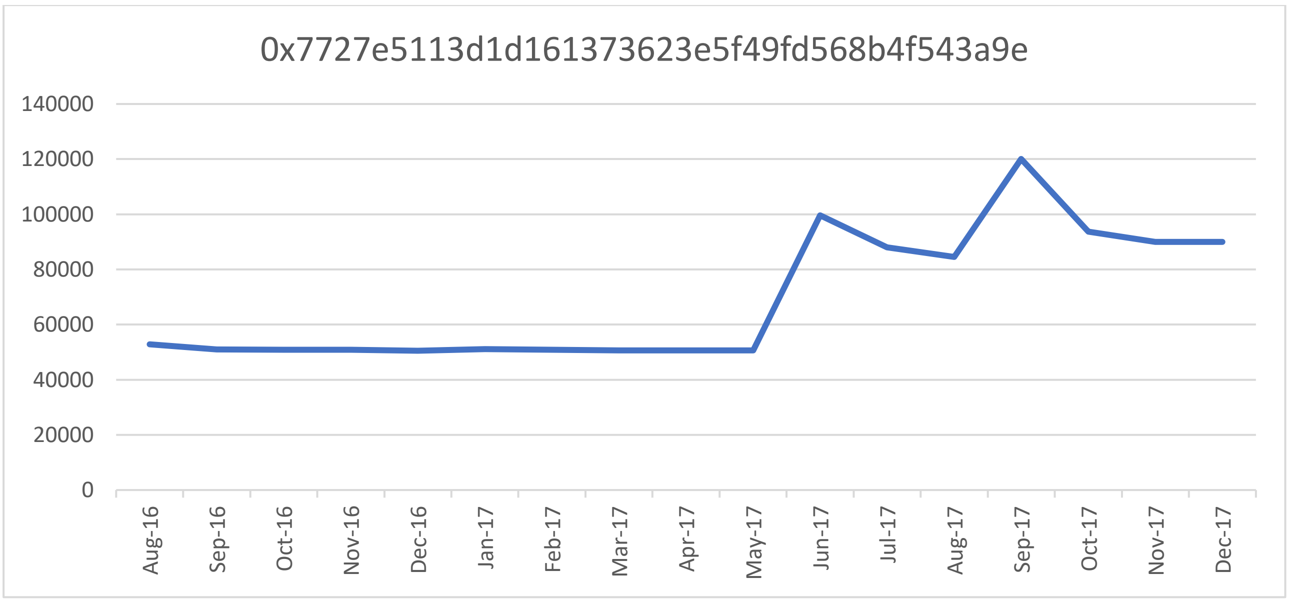
Result Analysis:

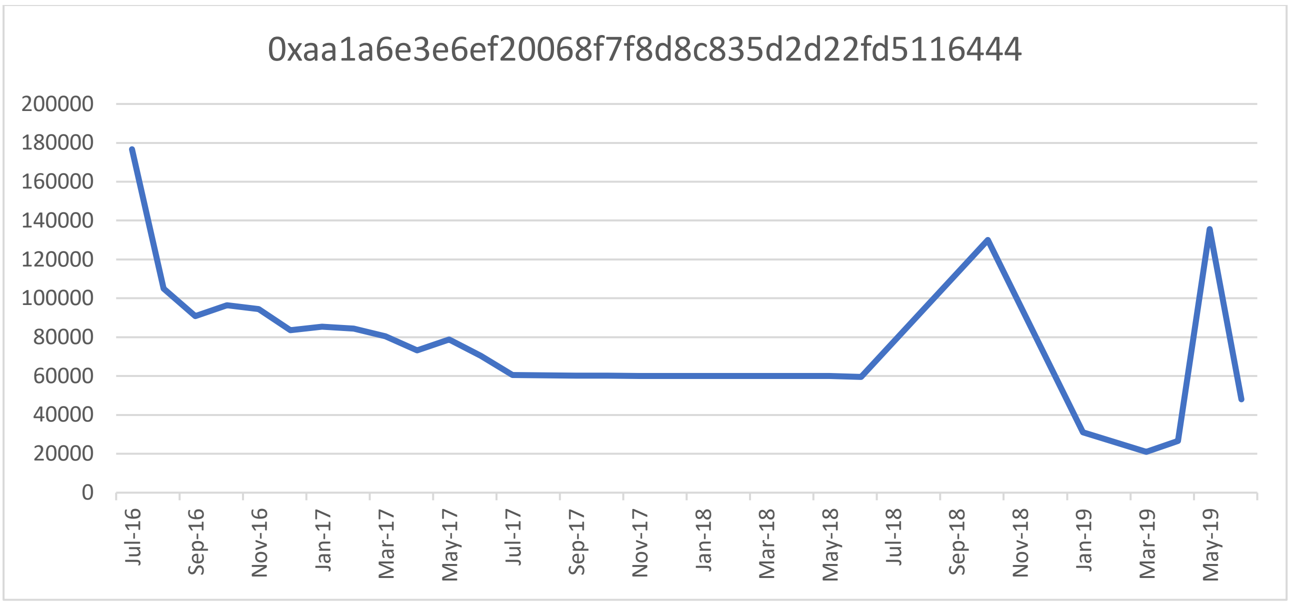
The final result is the average of gas consumed by the top 10 smart contracts per month from its start time to end time. Below are a series of graphs plotted using the data.

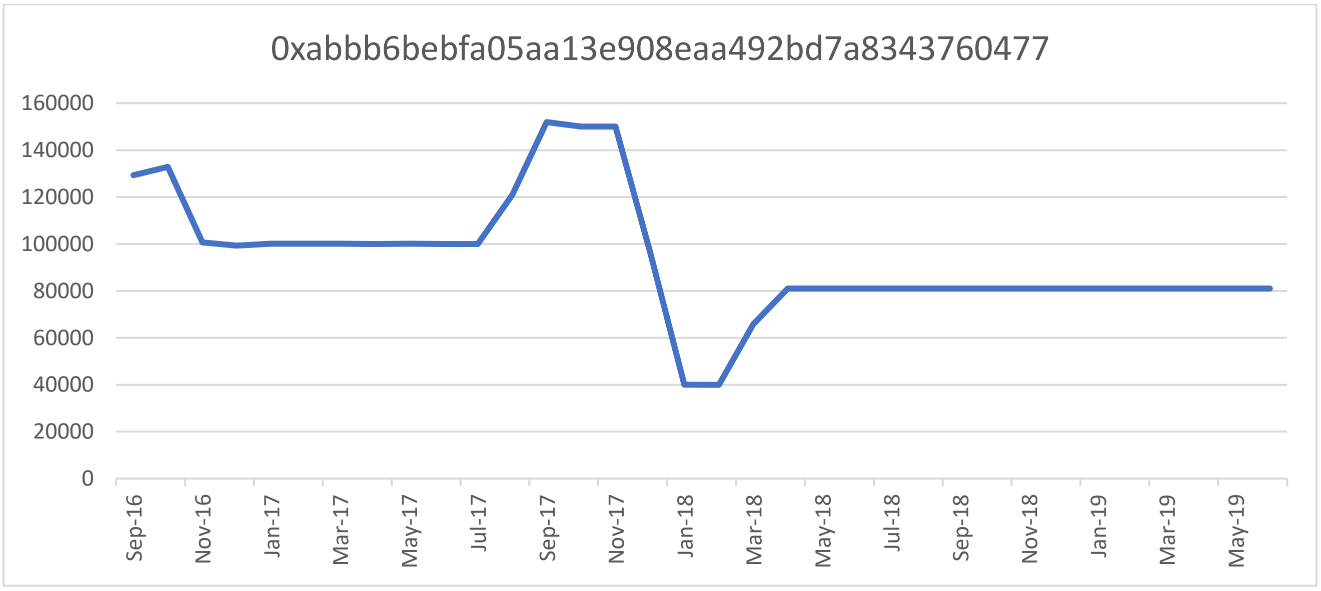


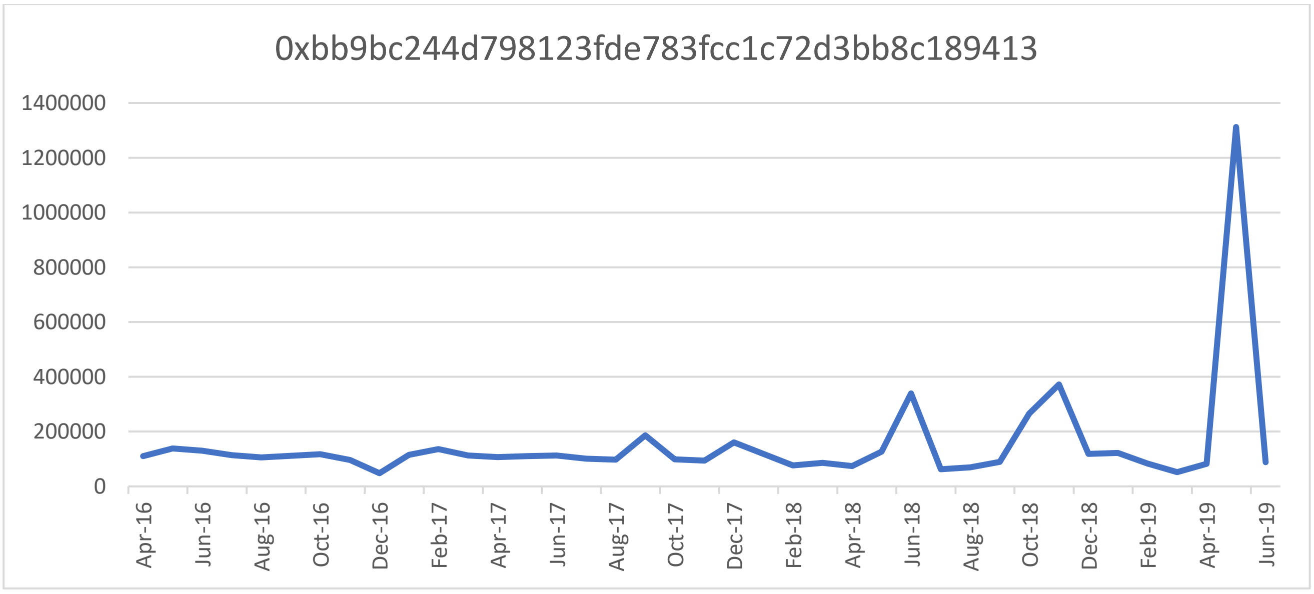

0x341e790174e3a4d35b65fdc067b6b5634a61caea
250000
200000
150000
100000
50000
0
Jul-16
Aug-16

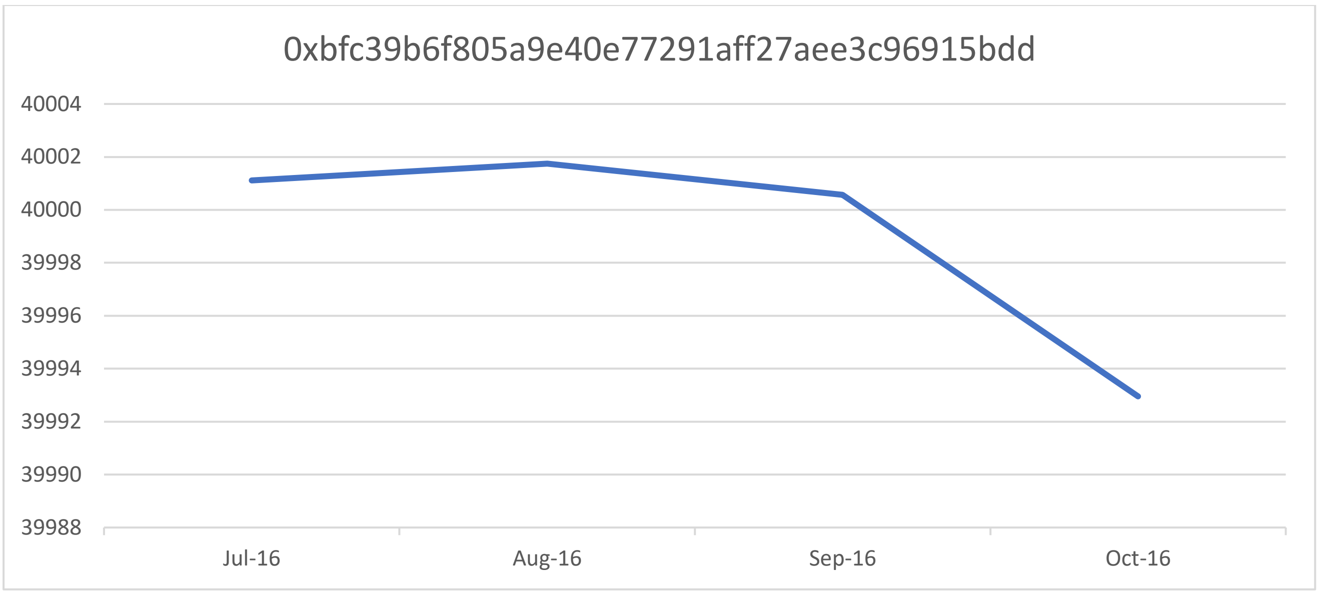

0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8
47400
47300
47200
47100
47000
46900
46800
Jul-16
Sep-16
Nov-16
Jan-17
Mar-17
May-17
Jul-17
Sep-17
Nov-17
Jan-18
Mar-18
May-18
Jul-18
Sep-18
Nov-18
Jan-19
Mar-19
May-19

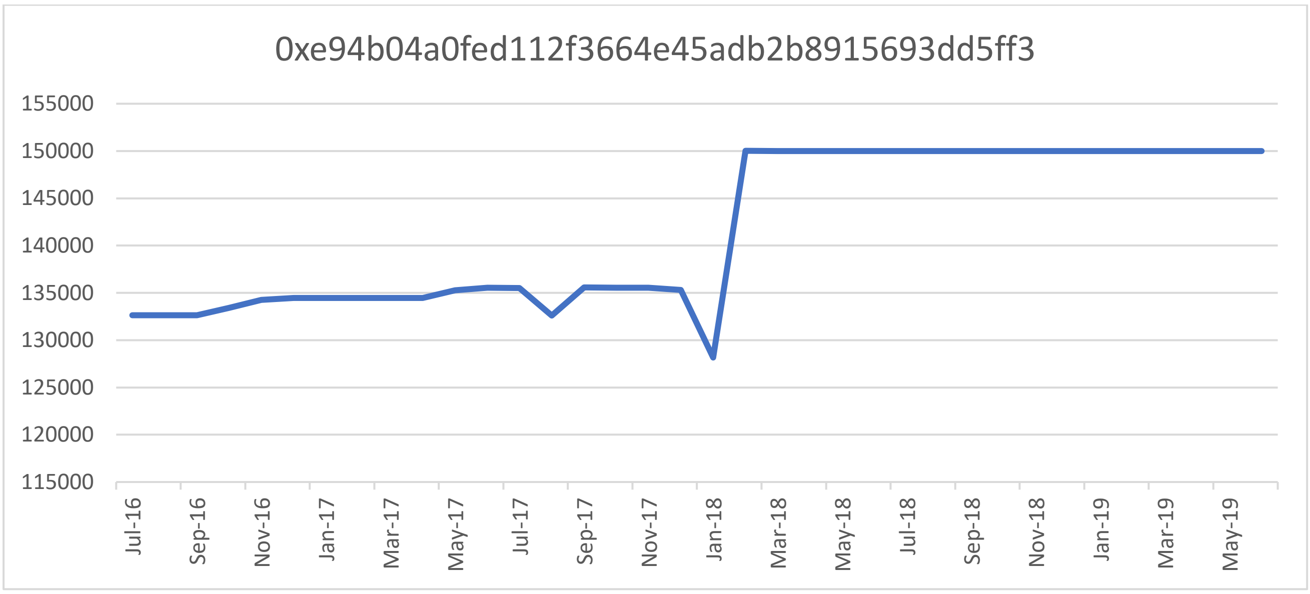

0x7727e5113d1d161373623e5f49fd568b4f543a9e
140000
120000
100000
80000
60000
40000
20000
0
Aug-16
Sep-16
Oct-16
Nov-16
Dec-16
Jan-17
Feb-17
Mar-17
Apr-17
May-17
Jun-17
Jul-17
Aug-17
Sep-17
Oct-17
Nov-17
Dec-17

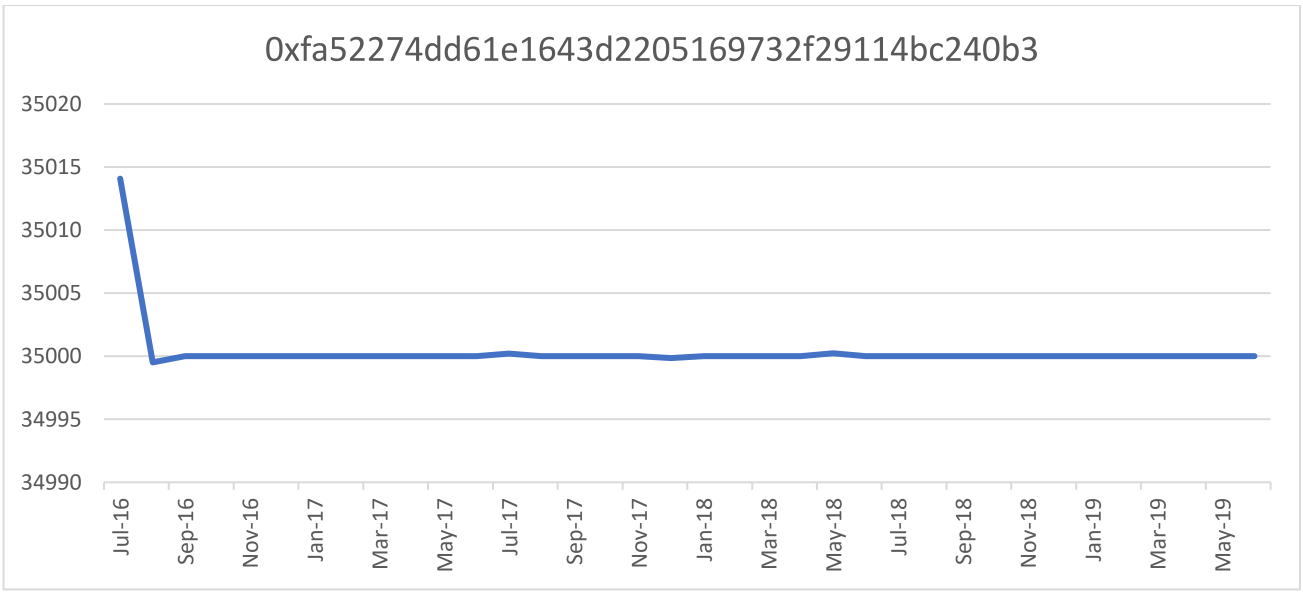

0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
Jul-16
Sep-16
Nov-16
Jan-17
Mar-17
May-17
Jul-17
Sep-17
Nov-17
Jan-18
Mar-18
May-18
Jul-18
Sep-18
Nov-18
Jan-19
Mar-19
May-19


0xabbb6bebfa05aa13e908eaa492bd7a8343760477
160000
140000
120000
100000
80000
60000
40000
20000
0
Sep-16
Nov-16
Jan-17
Mar-17
May-17
Jul-17
Sep-17
Nov-17
Jan-18
Mar-18
May-18
Jul-18
Sep-18
Nov-18
Jan-19
Mar-19
May-19


0xbb9bc244d798123fde783fcc1c72d3bb8c189413
1400000
1200000
1000000
800000
600000
400000
200000
0
Apr-16
Jun-16
Aug-16
Oct-16
Dec-16
Feb-17
Apr-17
Jun-17
Aug-17
Oct-17
Dec-17
Feb-18
Apr-18
Jun-18
Aug-18
Oct-18
Dec-18
Feb-19
Apr-19
Jun-19


0xbfc39b6f805a9e40e77291aff27aee3c96915bdd
40004
40002
40000
39998
39996
39994
39992
39990
39988
Jul-16
Aug-16
Sep-16
Oct-16


0xe94b04a0fed112f3664e45adb2b8915693dd5ff3
155000
150000
145000
140000
135000
130000
125000
120000
115000
Jul-16
Sep-16
Nov-16
Jan-17
Mar-17
May-17
Jul-17
Sep-17
Nov-17
Jan-18
Mar-18
May-18
Jul-18
Sep-18
Nov-18
Jan-19
Mar-19
May-19


0xfa52274dd61e1643d2205169732f29114bc240b3
35020
35015
35010
35005
35000
34995
34990
Jul-16
Sep-16
Nov-16
Jan-17
Mar-17
May-17
Jul-17
Sep-17
Nov-17
Jan-18
Mar-18
May-18
Jul-18
Sep-18
Nov-18
Jan-19
Mar-19
May-19

- The complexity of gas requirement by the top 10 smart contracts are shown in each graph. Few smart contracts require more gas over the time period but whereas few contracts require less gas.

SCAM ANALYSIS

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)

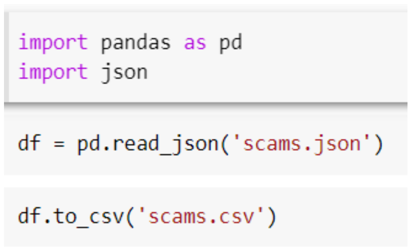
Part A: Most lucrative form of scam

Approach:

To identify the most lucrative form of scam I have joined the scams dataset and transactions dataset based on address of transaction as key. From the joined dataset the aggregate of values with scam category as key is calculated. From the aggregate output the scam category with highest values is the most lucrative form of scam.

Data Preprocessing:

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



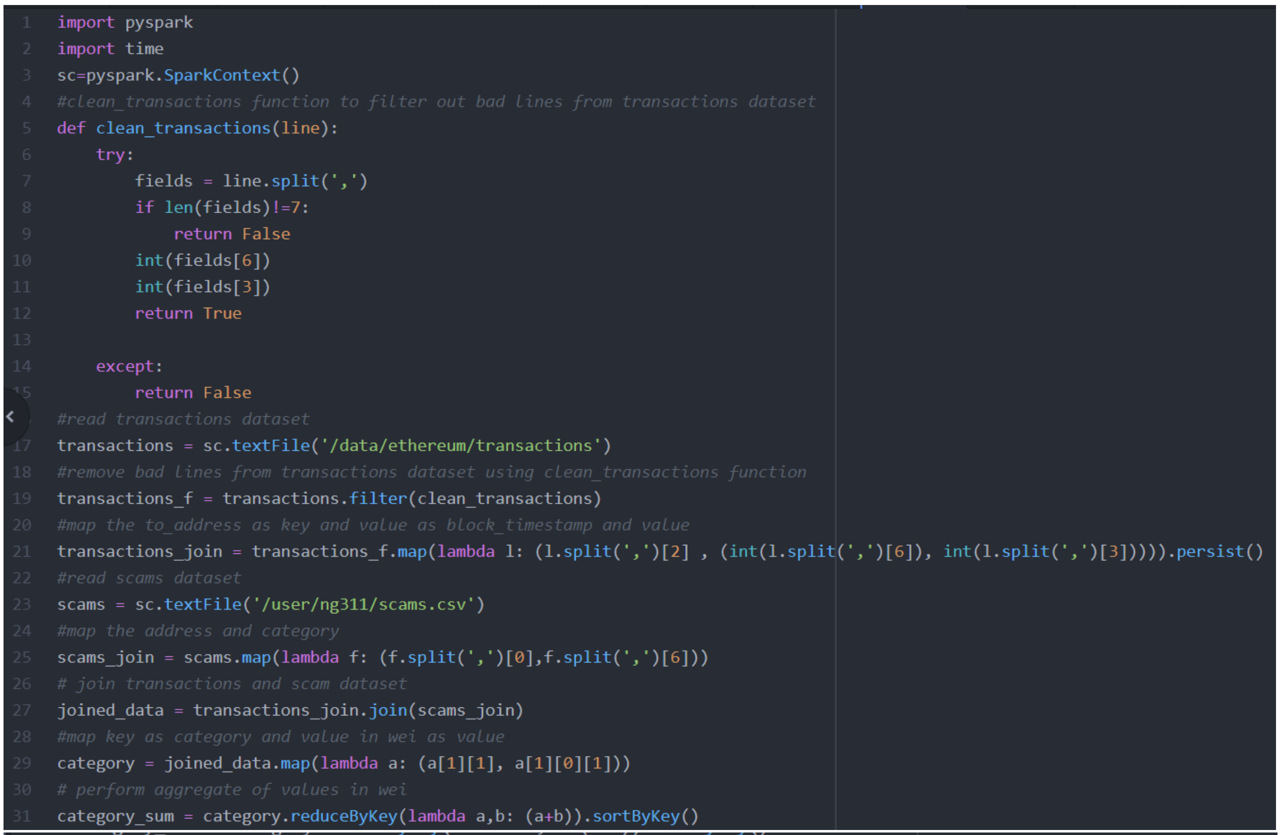
```
import pandas as pd
import json
```

```
df = pd.read_json('scams.json')
```

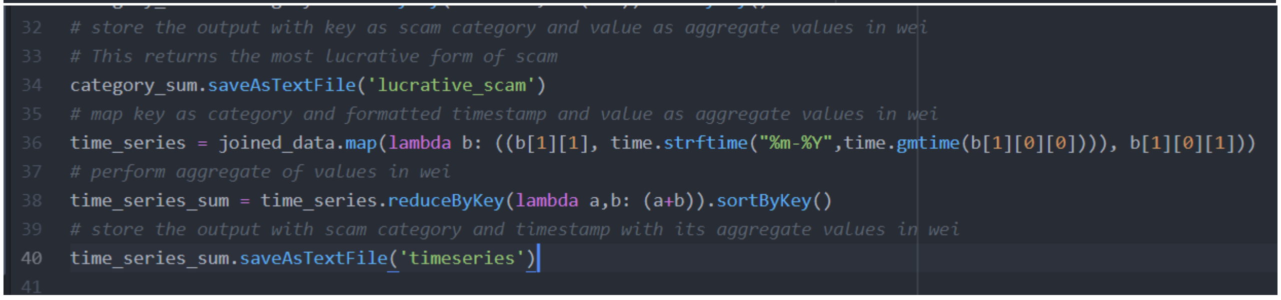
```
df.to_csv('scams.csv')
```

The converted CSV file is stored back to HDFS as **/user/ng311/scams.csv**.

Code: Program name is scamspark.py



```
import pyspark
import time
sc=pyspark.SparkContext()
#clean_transactions function to filter out bad lines from transactions dataset
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
#read transactions dataset
transactions = sc.textFile('/data/ethereum/transactions')
#remove bad lines from transactions dataset using clean_transactions function
transactions_f = transactions.filter(clean_transactions)
#map the to_address as key and value as block_timestamp and value
transactions_join = transactions_f.map(lambda l: (l.split(',')[2] , (int(l.split(',')[6]), int(l.split(',')[3])))).persist()
#read scams dataset
scams = sc.textFile('/user/ng311/scams.csv')
#map the address and category
scams_join = scams.map(lambda f: (f.split(',')[0],f.split(',')[6]))
# join transactions and scam dataset
joined_data = transactions_join.join(scams_join)
#map key as category and value in wei as value
category = joined_data.map(lambda a: (a[1][1], a[1][0][1]))
# perform aggregate of values in wei
category_sum = category.reduceByKey(lambda a,b: (a+b)).sortByKey()
```



```
# store the output with key as scam category and value as aggregate values in wei
# This returns the most lucrative form of scam
category_sum.saveAsTextFile('lucrative_scam')
# map key as category and formatted timestamp and value as aggregate values in wei
time_series = joined_data.map(lambda b: ((b[1][1], time.strftime("%m-%Y",time.gmtime(b[1][0][0]))), b[1][0][1]))
# perform aggregate of values in wei
time_series_sum = time_series.reduceByKey(lambda a,b: (a+b)).sortByKey()
# store the output with scam category and timestamp with its aggregate values in wei
time_series_sum.saveAsTextFile('timeseries')
```

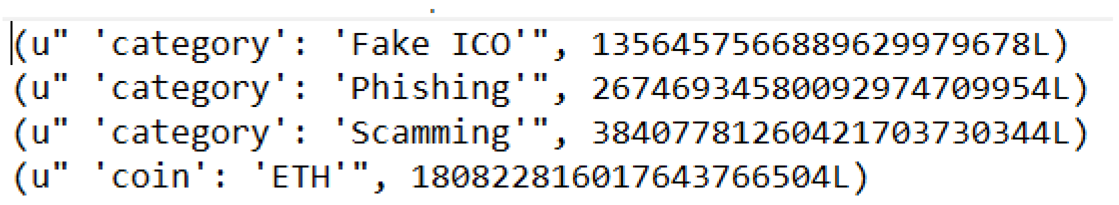
Code Explanation:

- Spark job to find the most lucrative form of scam. Transactions dataset and scams.csv dataset is used here.
- Transactions dataset is read. Using spark's map function we map the key as to_address and values as block_timestamp and values in wei.
- Next scams.csv dataset is read. Using spark's map function we map the key as address and value as category.
- The join operation is performed for both the datasets. From the joined dataset again map function is used to map the key as category and values in wei.
- Reduce by Key operation is performed to calculate the aggregate of values in wei. The result will be the spam categories with sum of values in wei. The output is stored in HDFS.
- In the output file the scam category with high values of wei is the most lucrative form of scam.

Spark Job ID -
http://andromeda.student.eecs.qmul.ac.uk:18088/history/application_1575381276332_2540/jobs/

Result Analysis:

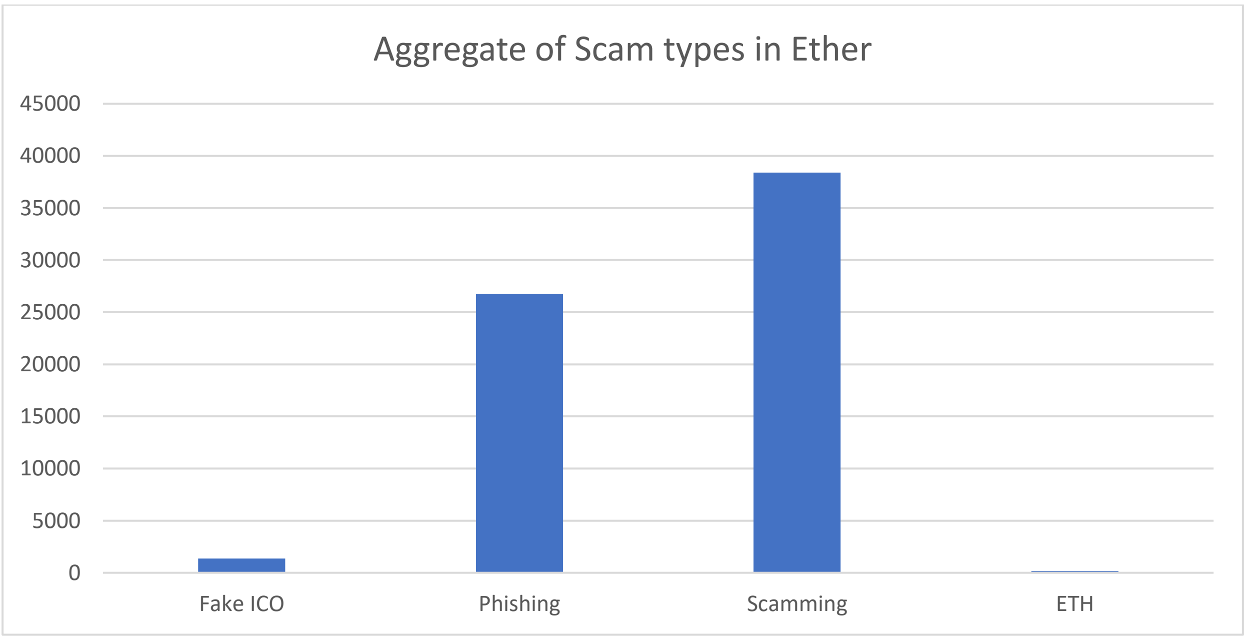
Below is the output,



```
(u" 'category': 'Fake ICO'", 1356457566889629979678L)
(u" 'category': 'Phishing'", 26746934580092974709954L)
(u" 'category': 'Scamming'", 38407781260421703730344L)
(u" 'coin': 'ETH'", 180822816017643766504L)
```

From the result the most lucrative form of scam is “Scamming” as the values transferred in Wei is the highest for this particular category of scam.

The corresponding plot in ether is shown below,

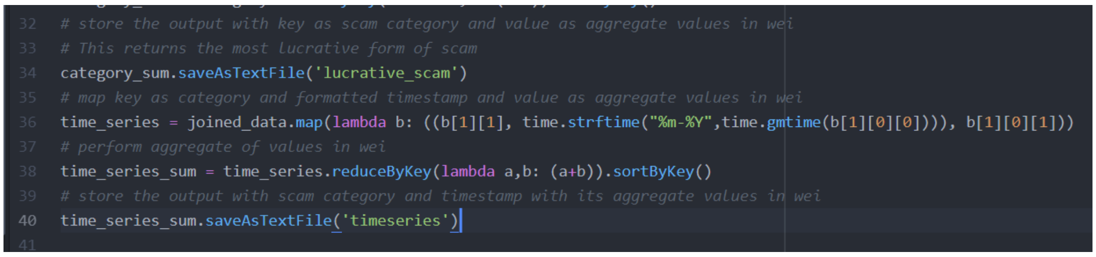


Part B: Time series analysis of different types of scams

Approach:

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

Code:



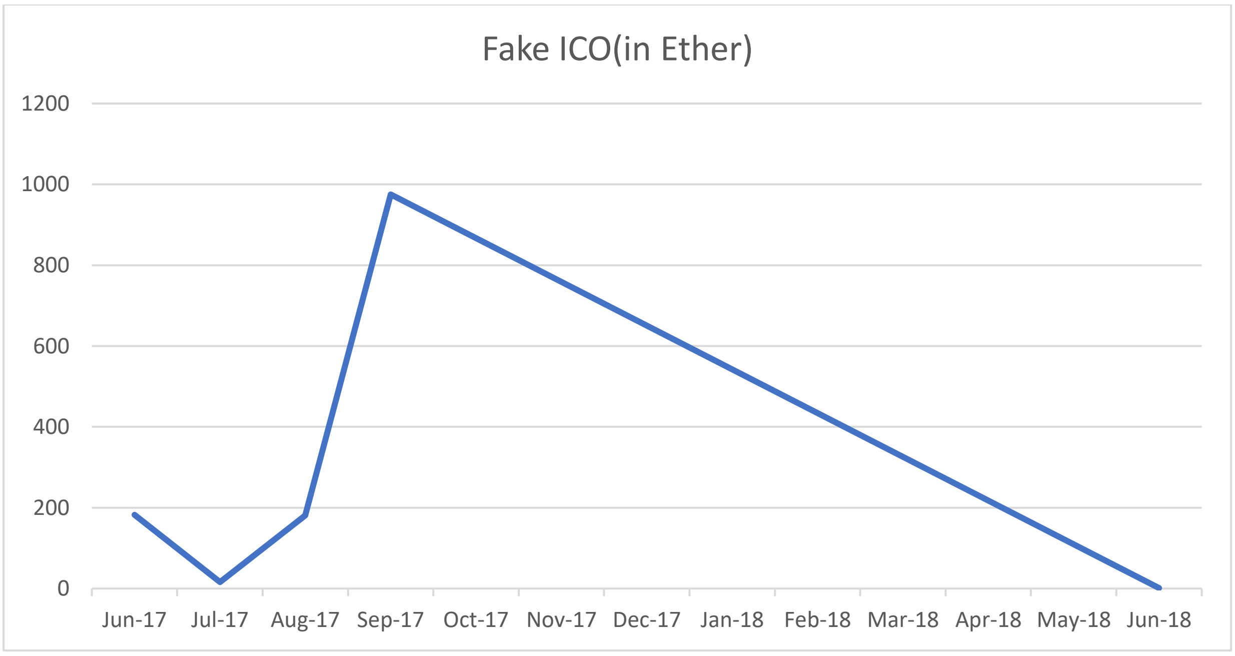
```
# store the output with key as scam category and value as aggregate values in wei
# This returns the most lucrative form of scam
category_sum.saveAsTextFile('lucrative_scam')
# map key as category and formatted timestamp and value as aggregate values in wei
time_series = joined_data.map(lambda b: ((b[1][1], time.strftime("%m-%Y",time.gmtime(b[1][0][0]))), b[1][0][1]))
# perform aggregate of values in wei
time_series_sum = time_series.reduceByKey(lambda a,b: (a+b)).sortByKey()
# store the output with scam category and timestamp with its aggregate values in wei
time_series_sum.saveAsTextFile('timeseries')
```

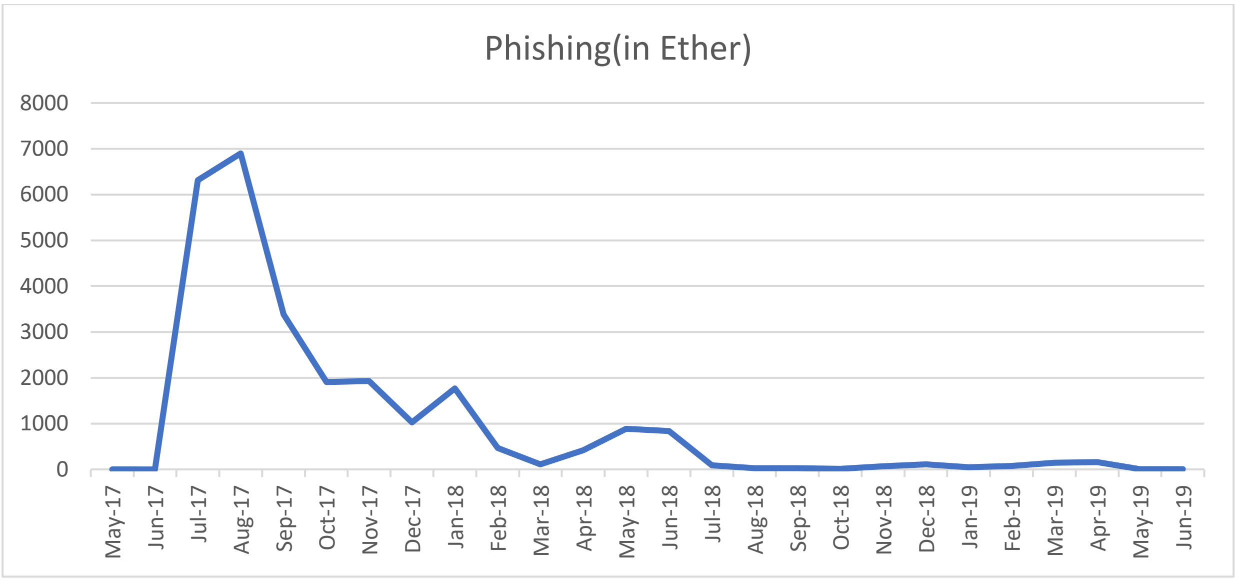
Code Explanation:

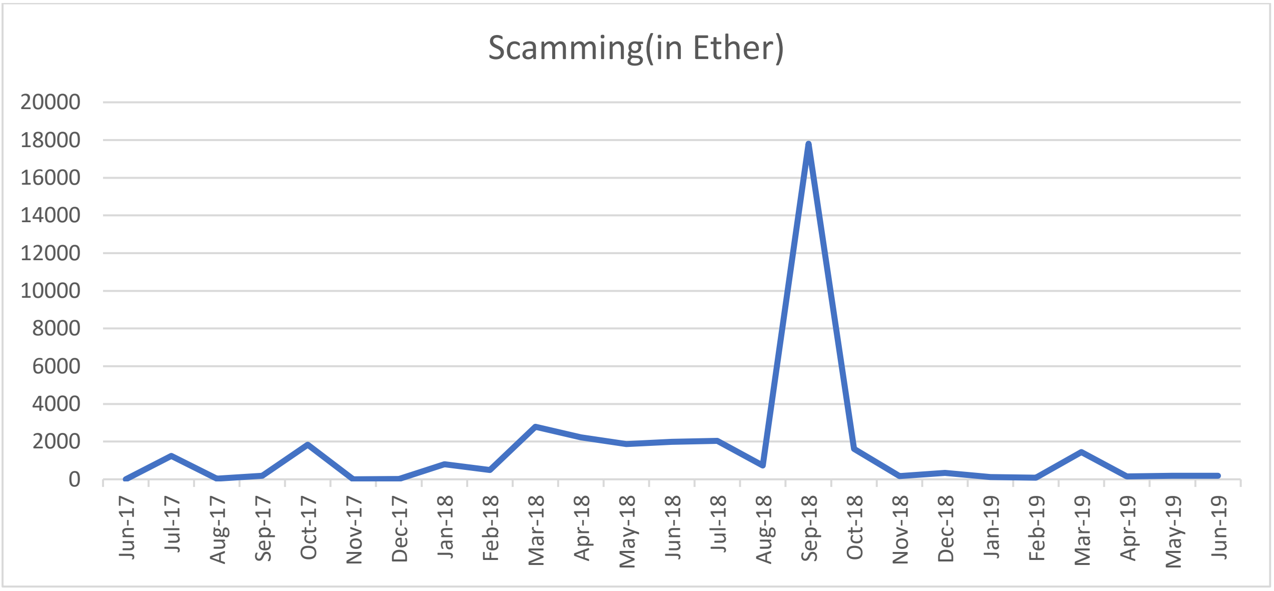
- From the joined dataset spark's map function is used to map the key as scam category and timestamp and value as values in wei. The Unix timestamp is converted to Gregorian format using time function.
- Spark's reduce by key method is used to calculate the aggregate of values in wei.
- The result of 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,







All the wei values are converted to ether and plotted. From the graph we can see that the most popular scam “Scamming” was at its peak during Sep-2018.

Report by,

Niranjan Ganesan

Student ID - 170963389