COURSEWORK: ETHEREUM ANALYSIS

PART A. TIME ANALYSIS (20%)
Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.
Create a bar plot showing the average value of transaction in each 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.)

DATASET SCHEMA – TRANSACTIONS

block_number	Block number where this transaction was in
from_address	Address of the sender
to_address	Address of the receiver. null when it is a contract creation transaction
value	Value transferred in Wei (the smallest denomination of ether)
gas	Gas provided by the sender
gas_price	Gas price provided by the sender in Wei
block_timestamp	Timestamp the associated block was registered at (effectively timestamp of the transaction)

- To calculate number of transactions occurring every month the **block_timestamp** field is used.
- The aggregate **sum(values)** of all the timestamp values per month returns the total number of transactions occurred.

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

class PartA(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if(len(fields)==7):
                time_epoch = int(fields[6])
                year = time.strftime("%Y-%m", time.gmtime(time_epoch))
                yield(year, 1)

        except:
            pass

    def combiner(self, keys, values):
        yield(keys, sum(values))

    def reducer(self, keys, values):
        yield(keys, sum(values))

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

Figure 1: PartA.py

Code Explanation:

- Libraries imported are MRJob and time.
- Map/Reduce job to calculate initial aggregation reads the lines in the **transactionSmall** file with delimiter comma.
- The if statement checks if the **len(fields)** is exactly equal to the number of columns in the dataset **(7).**
- The block_timestamp field is defined as **int** to process the timestamp values. The **time.gmtime** is used to convert the Unix timestamp values to Gregorian format MM-YY.
- The mapper yields the formatted timestamp values.
- **sum(values)** in the **combiner** performs the aggregation on timestamp value which returns the total count of transaction in the mapper.
- **sum(values)** in the **reducer** calculates the aggregate of the transaction values.
- The final output yields the total count of transaction per month.

```
partdscam.py
"2015-08" 85609
"2015-11" 234733
"2016-01" 404816
"2016-03" 917170
"2016-05" 1346796
"2016-07" 1356907
"2016-09" 1387412
"2016-10" 1329847
"2016-12" 1316131
"2017-02" 1410048
"2017-04" 2539966
"2017-06" 7244657
"2015-09" 173805
"2015-10" 205045
"2015-12" 347092
"2016-02" 520040
"2016-04" 1023096
"2016-06" 1351536
"2016-08" 1405743
"2016-11" 1301586
"2017-01" 1409664
"2017-03" 2426471
"2017-05" 4245516
"2017-07" 3536296
```

Figure 3: PartA_out

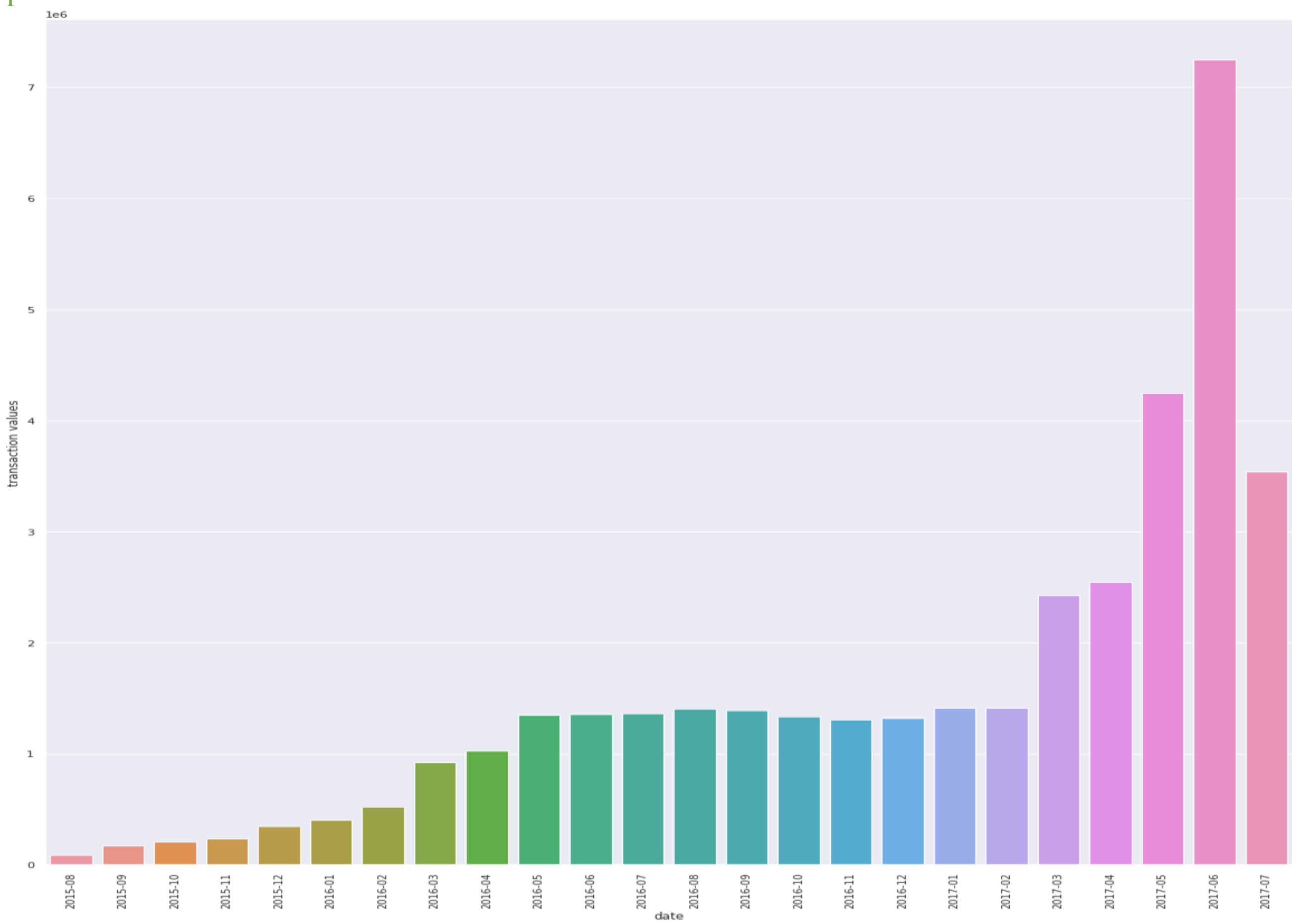
Command Line:

>>python PartA.py -r hadoop –output-dir PartAout –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactionSmall

Job Id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_7787/

Graph:



Graph 1: Number of Transaction occurring every month

- To calculate the average value of transactions each month we use **value** fields.

Gasguzzler.py Gasguzzler_out.txt PartA_avg.py PartB_spark.py

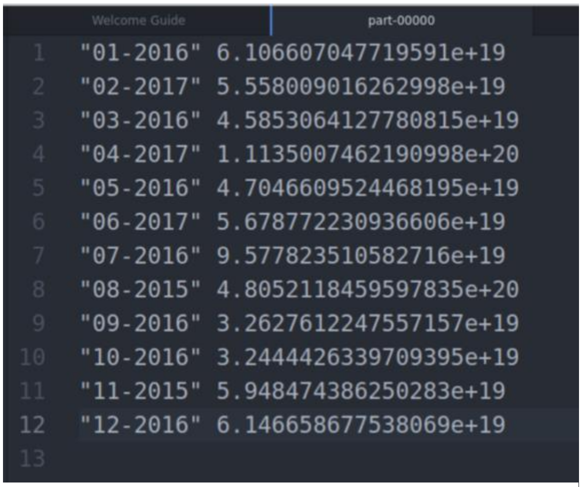
```python
from mrjob.job import MRJob
import time

class PartA(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if(len(fields)==7):
                time_epoch = int(fields[6])
                monthYear = time.strftime("%m-%Y", time.gmtime(time_epoch))
                yield((monthYear), (int(fields[3]), 1))

        except:
            pass

    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, (total, count))

    def reducer(self, feature, values):
        count = 0
        total = 0
```

```python
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, (total, count))

    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, total/count)

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

Figure 3: PartA_avg.py

Code Explanation:

- A Map/Reduce job is used to perform this computation. In mapper, **key** is **month-year** in Gregorian format and values are **value** and **count 1** to count the number of occurrences which is used later in reducer to calculate the average.
- In combiner, the key is the formatted timestamp and values are value and count. A for loop is used to calculate the sum of values and its number of occurrences per month.
- In reducer, the total/count yields average values per month. The average is calculated by dividing the sum of values with its total number of occurrences.
- Finally, the result is the average value of transaction per month.



```
Welcome Guide    part-00000
"01-2016" 6.106607047719591e+19
"02-2017" 5.558009016262998e+19
"03-2016" 4.5853064127780815e+19
"04-2017" 1.1135007462190998e+20
"05-2016" 4.7046609524468195e+19
"06-2017" 5.678772230936606e+19
"07-2016" 9.577823510582716e+19
"08-2015" 4.8052118459597835e+20
"09-2016" 3.2627612247557157e+19
"10-2016" 3.2444426339709395e+19
"11-2015" 5.948474386250283e+19
"12-2016" 6.146658677538069e+19
```

Figure 4: PartA_avg_out

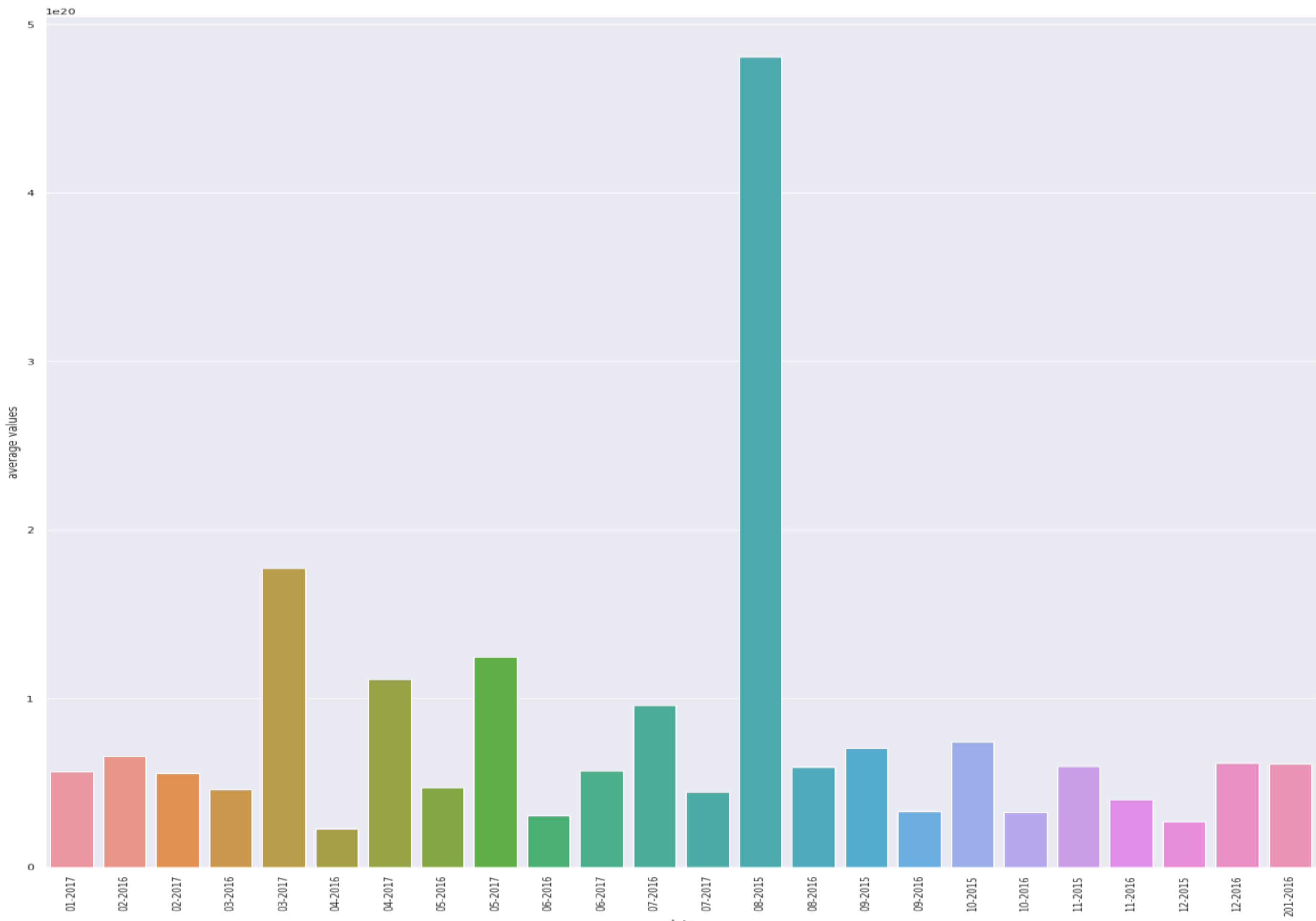
Command Line:

>>python PartA_avg.py -r hadoop –output-dir PartA_avg_out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactionSmall

Job Id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1606730688641_5889/

Graph:



Graph 2: Average Value of transaction each month

PART B. TOP TEN MOST POPULAR SERVICES (20%)

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.

DATASET SCHEMA - CONTRACTS

address	Address of the contract
is_erc20	Whether this contract is an ERC20 contract
is_erc721	Whether this contract is an ERC721 contract
block_number	Block number where this contract was created

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.

PartB_1.py

```
from mrjob.job import MRJob
class PartB_Job1(MRJob):

  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

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

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

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

Figure 5: PartB_1.py

Code Explanation:

- Map/Reduce job is used to calculate the initial aggregation with key as **to_address** and value as **values** from transaction file.
- 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 improves 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.

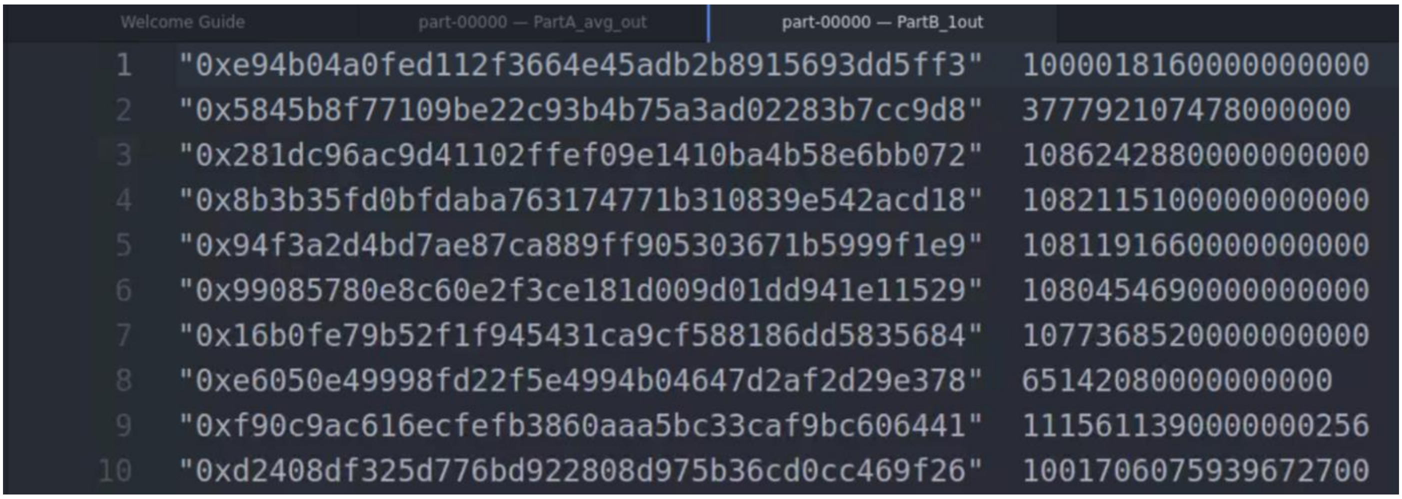
Command Line:

>>python PartB_1.py -r hadoop –output-dir PartB_1out –no-cat-output
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactionSmall

Job Id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1075/

Output Snippet:



```
Welcome Guide    part-00000 — PartA_avg_out    part-00000 — PartB_1out
1  "0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"   1000018160000000000
2  "0x5845b8f77109be22c93b4b75a3ad02283b7cc9d8"   377792107478000000
3  "0x281dc96ac9d41102ffef09e1410ba4b58e6bb072"   1086242880000000000
4  "0x8b3b35fd0bfdaba763174771b310839e542acd18"   1082115100000000000
5  "0x94f3a2d4bd7ae87ca889ff905303671b5999f1e9"   1081191660000000000
6  "0x99085780e8c60e2f3ce181d009d01dd941e11529"   1080454690000000000
7  "0x16b0fe79b52f1f945431ca9cf588186dd5835684"   1077368520000000000
8  "0xe6050e49998fd22f5e4994b04647d2af2d29e378"   651420800000000000
9  "0xf90c9ac616ecfefb3860aaa5bc33caf9bc606441"   1115611390000000256
10 "0xd2408df325d776bd922808d975b36cd0cc469f26"   1001706075939672700
```

Figure 6: PartB_1out

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.

- We perform repartition join between **contracts** dataset and t**ransactionSmall** dataset (JOB 1 output) to filter out user address from smart contract address. The output will only list address that belongs to smart contracts.

PartB_2.py

```python
from mrjob.job import MRJob

class PartB_Job2(MRJob):

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

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

        except:
            pass

    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()
```

Figure 7: PartB_2.py

Code Explanation:

- Map/Reduce job to perform repartition join between contracts dataset and transaction aggregate dataset (PartB_1_out.txt).
- 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.

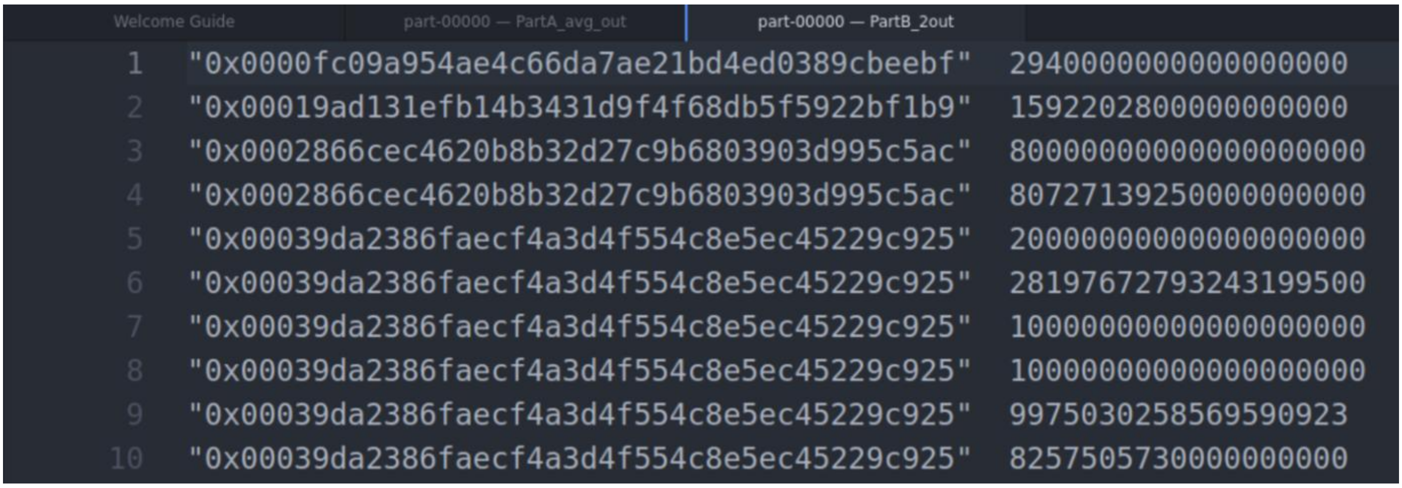
Command Line:

>>python PartB_2.py -r hadoop –output-dir PartB_2_out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts PartB_1_out.txt

Job Id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1084/

Output Snippet:




```
"0x0000fc09a954ae4c66da7ae21bd4ed0389cbeebf"   2940000000000000000
"0x00019ad131efb14b3431d9f4f68db5f5922bf1b9"   1592202800000000000
"0x0002866cec4620b8b32d27c9b6803903d995c5ac"   80000000000000000000
"0x0002866cec4620b8b32d27c9b6803903d995c5ac"   80727139250000000000
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"   20000000000000000000
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"   28197672793243199500
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"   10000000000000000000
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"   10000000000000000000
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"   9975030258569590923
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"   8257505730000000000
```

Figure 8: PartB_2out

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.

- The output from the previous JOB 2 will be taken as input in JOB3.
- The values will be sorted based on total aggregate value to obtain the top 10 contracts.

```python
from mrjob.job import MRJob

class PartB_Job3(MRJob):

    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

    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

    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()
```

Figure 9: PartB_3.py

Code Explanation:

- Map/Reduce job is 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 sorted 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.

Job Execution:

```
(Coursework) abs01@itl220 ~/ECS765/Coursework> python PartB_3.py PartB_2out.txt > PartB_3out.txt
Using configs in /homes/abs01/.mrjob.conf
No configs specified for inline runner
Creating temp directory /tmp/PartB_3.abs01.20201210.185616.267592
Running step 1 of 1...
job output is in /tmp/PartB_3.abs01.20201210.185616.267592/output
Streaming final output from /tmp/PartB_3.abs01.20201210.185616.267592/output...
Removing temp directory /tmp/PartB_3.abs01.20201210.185616.267592...
```

Figure 10: PartB_3run

Output:

```
PartB_3out.txt
1 "0xbfc39b6f805a9e40e77291aff27aee3c96915bd - 100000000000000000000000"
2 "0x52965f9bd9d0f2bbea9b5a9c155a455d0e58fe2 - 92573218438222657658190 9"
3 "0x209c4784ab1e8183cf58ca33cb740efbf3fc18e - 84999999939168000000000"
4 "0xbfc39b6f805a9e40e77291aff27aee3c96915bd - 79999999913194000000000"
5 "0xbfc39b6f805a9e40e77291aff27aee3c96915bd - 70371324912610000000000"
6 "0xab7c74abc0c4d48d1bdad5dcb26153fc8780f83 - 70000000000000000000000"
7 "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd511644 - 67268461248232864470761 4"
8 "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd511644 - 67242435428759845676138 6"
9 "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f - 61110201445604000000000"
10  "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd511644 - 55566000000000000000000"
```

Figure 11: PartB_3out

PART C. TOP TEN MOST ACTIVE MINERS (10%)

Evaluate the top 10 miners by the size of the blocks mined. This is simpler as it does not require a join. You will first have to aggregate blocks to see how much each miner has been involved in. You will want to aggregate size for addresses in the miner field. This will be similar to the wordcount that we saw in Lab 1 and Lab 2. You can add each value from the reducer to a list and then sort the list to obtain the most active miners.

DATASET SCHEMA – BLOCKS

number	The block number
hash	Hash of the block
miner	The address of the beneficiary to whom the mining rewards were given
difficulty	Integer of the difficulty for this block
size	The size of this block in bytes
gas_limit	The maximum gas allowed in this block
gas_used	The total used gas by all transactions in this block
timestamp	The timestamp for when the block was collated
transaction_count	The number of transactions in the block

- We perform initial aggregation on **blocks** dataset.

PartC_1.py

```
from mrjob.job import MRJob
class PartC_Job1(MRJob):

  def mapper(self, _, line):
    try:
      fields = line.split(',')
      if len(fields)==9:
          miner = fields[2]
          count = int(fields[4])
          if count == 0:
              pass
          else:
              yield(miner,count)

    except:
        pass

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

    def reducer(self, address, count):
        yield(miner, sum(count))

if __name__ == '__main__':
    PartC_Job1.run()
```

Figure 12: PartC_1.py

Code Explanation:

- Map/Reduce job is used to calculate the initial aggregation with key as **miner** and value as **size**.
- 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 block mined present in each mapper.
- Adding a combiner improves the aggregation performance of the job.
- Finally, in the reducer the same sum operation is used to calculate the aggregate of blocks.
- The output is the sum of values of blocks mined. This tells us how much each miner has been involved in.

Command Line:

>>python PartC_1.py -r hadoop –output-dir PartC_1out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/blocks

Job Id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_1105/

Output Snippet:

```
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"   9773
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"   15532
"0x829bd824b016326a401d083b33d092293333a830"   14033
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"   29386
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"   28954
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"   21030
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"   14168
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"   29125
"0x829bd824b016326a401d083b33d092293333a830"   27294
"0xea674fdde714fd979de3edf0f56aa9716b898ec8"   17090
```

Figure 13: PartC_1out

- The output of the **PartC_1** is used as an input to obtain the top 10 miners.

PartC_2.py

```
from mrjob.job import MRJob

class PartC_Job2(MRJob):

    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

    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

    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__':
    PartC_Job2.run()
```

Figure 14: PartC_2.py

Code Explanation:

- Map/Reduce job is to filter out top 10 miners obtained as a result from Job 1 output. (PartC_1out.txt)
- 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 sorted 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 miners.

Job Execution:

```
(Coursework) abs01@itl220 ~/ECS765/Coursework> python PartC_2.py PartC_1out.txt > PartC_2out.txt
Using configs in /homes/abs01/.mrjob.conf
No configs specified for inline runner
Creating temp directory /tmp/PartC_2.abs01.20201210.190331.360887
Running step 1 of 1...
job output is in /tmp/PartC_2.abs01.20201210.190331.360887/output
Streaming final output from /tmp/PartC_2.abs01.20201210.190331.360887/output...
Removing temp directory /tmp/PartC_2.abs01.20201210.190331.360887...
```

Figure 15: PartC_2run

Output:

```
PartC_2out.txt
1 "0xea674fdde714fd979de3edf0f56aa9716b898ec - 718140"
2 "0xdc3f366882d53c6d5eb808018acfd1cfaa7ee45 - 525527"
3 "0x6c7f03ddfdd8a37ca267c88630a4fee958591de - 353029"
4 "0x1e9939daaad6924ad004c2560e9080416490034 - 331610"
5 "0x4bb96091ee9d802ed039c4d1a5f6216f90f81b0 - 331380"
6 "0x2a65aca4d5fc5b5c859090a6c34d16413539822 - 331030"
7 "0x2a65aca4d5fc5b5c859090a6c34d16413539822 - 330960"
8 "0xea674fdde714fd979de3edf0f56aa9716b898ec - 330856"
9 "0x2a65aca4d5fc5b5c859090a6c34d16413539822 - 330851"
10  "0x96338149e9f6c262d4cb7aeec1cf4c652079a11 - 330839"
```

Figure 16: PartC_2out

PART D. DATA EXPLORATION (50%)

The final part of the coursework requires you to explore the data and perform some analysis of your choosing. These tasks may be completed in either MRJob or Spark, and you may make use of Spark libraries such as MLlib (for machine learning) and GraphX for graph analysis. Below are some suggested ideas for analysis which could be undertaken, along with an expected grade for completing it to a goodstandard. You may attempt several of these tasks or undertake your own. However, it is recommended to discuss ideas with Joseph before commencing with them.

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? (15/50)

Most lucrative Scam Analysis:

Telemetry Consent | Forecast.py | partdscam.py | Mostlucrative.py | Scam_JobId

```
from mrjob.job import MRJob
from mrjob.step import MRStep
import json
import time

class partdscams(MRJob):
    scams = {}
    def mapper_join_init(self):
        with open("scams.json") as f:
            lineJSON = json.load(f)
            innerLines = lineJSON["result"]
            for line in innerLines:
                keyvals =  innerLines[line]
                addresses = keyvals["addresses"]
                id = keyvals["id"]
                name = keyvals["name"]
                category = keyvals["category"]
                try:
                    subcategory = keyvals["subcategory"]
                except:
                    subcategory = ""
                    continue
                status = keyvals["status"]
                id_list = [id,name,category,subcategory,status]
                for address in addresses:
                    id_list.append(address)
                    self.scams[address] = id_list
```

```python

    def mapper_repl_join(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 7:
                address = fields[2]
                value = int(fields[3])
                time_epoch = int(fields[6])
                if address in self.scams:
                   scam_details = self.scams[address]
                   yield (scam_details,value)
        except:
            pass

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

    def combiner_sum(self, key, values):
        scam_sum = sum(values)
        yield (key,scam_sum)

    def reducer_sum(self, key, values):
        scam_sum = sum(values)
        yield (None, (key,scam_sum))

    def mapper_sort(self,_,values):
        yield (None,values)
```

```python
    def combiner_sort(self,_,values):
        counter = 0
        sorted_val = sorted(values,reverse=True,key = lambda x:x[1])
        for topscam in sorted_val:
            yield (None,topscam)
            counter+=1
            if counter > 0:
                break

    def reducer_sort(self,_,values):
        counter = 0
        sorted_val = sorted(values,reverse=True,key = lambda x:x[1])
        for topscam in sorted_val:
            yield (None,topscam)
            counter+=1
            if counter > 0:
                break

    def steps(self):
        return [MRStep(mapper_init=self.mapper_join_init,mapper=self.mapper_repl_join),
        MRStep(mapper=self.mapper_length,combiner = self.combiner_sum,reducer=self.reducer_sum),
        MRStep(mapper=self.mapper_sort,combiner = self.combiner_sort,reducer=self.reducer_sort)]



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

Figure 17: Mostlucrative.py

Code Explanation:

- Imports MRJob, Json, time to implement the functions and obtain top 10 most lucrative scams.
- In **mapper_join_init**, inputs from **scams.json** and transactions are collected and all attributes are extracted.
- **mapper_replication_join** is used to fetch the values with respect to date.
- **mapper_length** will take input value and date and returns address, value and year.
- **combiner_sum** will take input as address, value and year from mapper_length then returns address, year and aggregate values. combiner is used to reduce the execution time in aggregating values.
- **reducer_sum** will perform same function as combiner_sum and returns address, year and aggregate values.
- **mapper_sort** takes input from reducer_sum and then return the values with respect to date.
- **combiner_sort** is takes input from mapper_sort and then sorts the values in ascending order. Finally returns top 10 scams.
- Finally, in reducer_sort same combiner operation is done in the reducer, where key value pairs are sorted again from all the combiners to achieve the same result. To iterate between all records from the combiner, A for loop is used and the result is shown in the top 10 order. The for loop is terminated if the iteration count exceeds 11 in order to show only the top 10 values.
- **mapping_steps** defines each mapper, combiner and reducer process flow.

Command Line:

>>python Mostlucrative.py -r hadoop –file hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/scams.json –output-dir Mostlucrative_out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactionSmall

Job Id:

Step1 – http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_6094/
Step2 – http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_6103/
Step3 – http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_6112/

Output:

Figure 18: Mostlucrative_out

Scam Analysis:

```python
from mrjob.job import MRJob
from mrjob.step import MRStep
import json
import time

class partdscams(MRJob):
    scams = {}
    def mapper_join_init(self):
        with open("scams.json") as f:
            lineJSON = json.load(f)
            innerLines = lineJSON["result"]
            for line in innerLines:
                keyvals =  innerLines[line]
                addresses = keyvals["addresses"]
                id = keyvals["id"]
                name = keyvals["name"]
                category = keyvals["category"]
                try:
                    subcategory = keyvals["subcategory"]
                except:
                    subcategory = ""
                    continue
                status = keyvals["status"]
                id_list = [id,name,category,subcategory,status]
                for address in addresses:
                    id_list.append(address)
                    self.scams[address] = id_list
```

```python

    def mapper_repl_join(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 7:
                address = fields[2]
                value = int(fields[3])
                time_epoch = int(fields[6])
                yearmonth = time.strftime("%Y %b", time.gmtime(time_epoch))
                if address in self.scams:
                   scam_details = self.scams[address]
                   yield (scam_details,(value,yearmonth))
        except:
            pass

    def mapper_length(self, key, value):
        yield ((key,value[1]), value[0])

    def combiner_sum(self, key, values):
        scam_sum = sum(values)
        yield (key,scam_sum)

    def reducer_sum(self, key, values):
        scam_sum = sum(values)
        yield (key,scam_sum)

    def mapper_sort(self,key,values):
```

```python
        newkey = key[1]
        yield (newkey,(values,key[0]))

    def combiner_sort(self,key,values):
        counter = 0
        sorted_val = sorted(values,reverse=True,key = lambda x:x[0])
        for topscam in sorted_val:
            yield (key,topscam)
            counter+=1
            if counter > 0:
                break
    def reducer_sort(self,key,values):
        counter = 0
        sorted_val = sorted(values,reverse=True,key = lambda x:x[0])
        for topscam in sorted_val:
            yield (key,topscam)
            counter+=1
            if counter > 0:
                break

    def steps(self):
        return [MRStep(mapper_init=self.mapper_join_init,mapper=self.mapper_repl_join),
        MRStep(mapper=self.mapper_length,combiner = self.combiner_sum,reducer=self.reducer_sum),
        MRStep(mapper=self.mapper_sort,combiner = self.combiner_sort,reducer=self.reducer_sort)]

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

Figure 19: partdscam.py

Command Line:

>>python partdscam.py -r hadoop –file hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/scams.json –output-dir ScamAnalysis_out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactionSmall

Job Id:

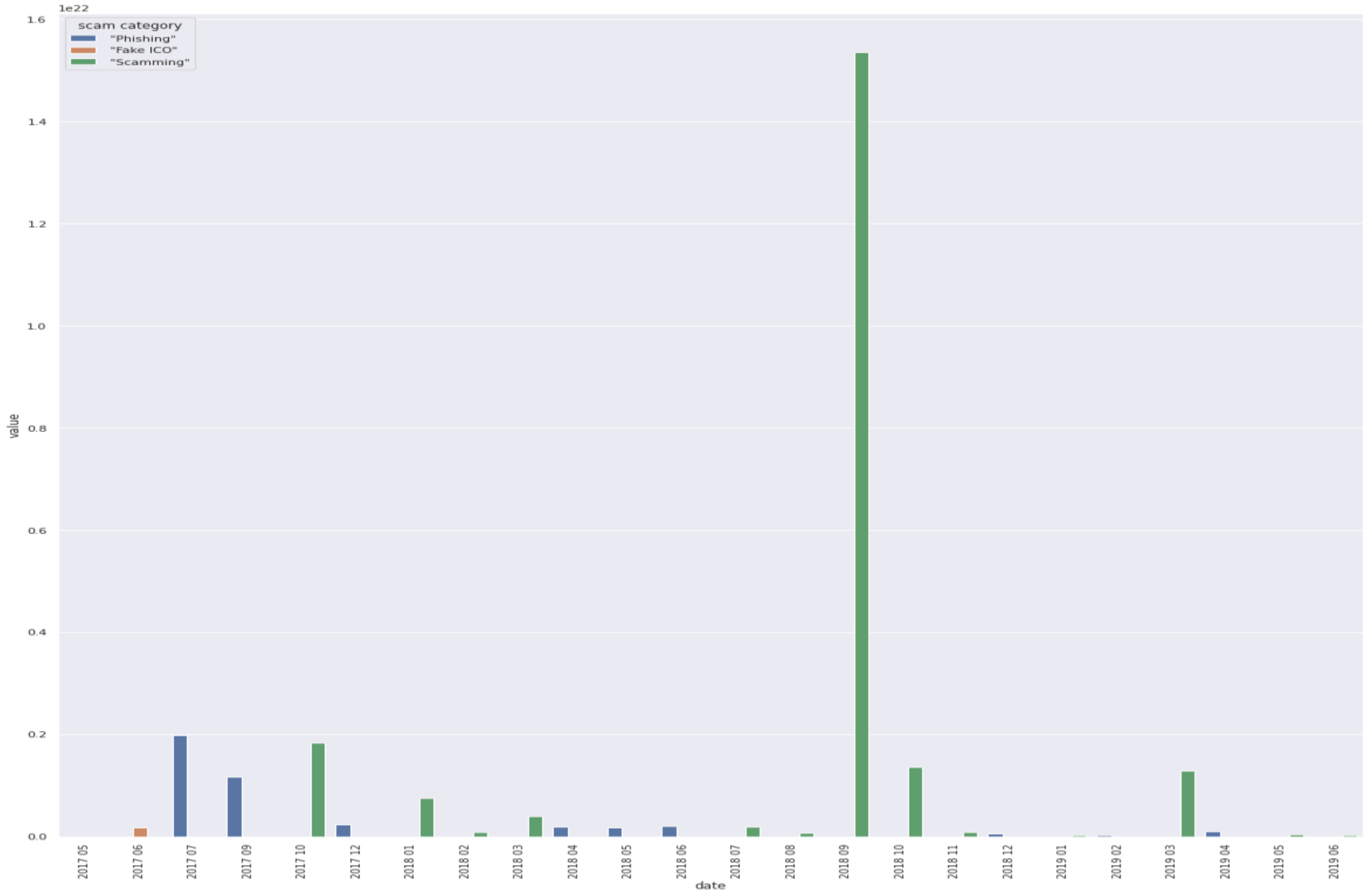
Step1 – http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_5358/
Step2 – http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_5368/
Step3 – http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_5373/

Output:

```
Welcome Guide    part-00000
"2017 Aug"  [2205893341814237414543, [2135, "OmiseGO.com", "Phishing", "OmiseGO", "Offline", "0x89c98cc6d9917b615257e5704e83906402f0f
"2017 Jul"  [1990536733530806579663, [90, "myetherwallet.tech", "Phishing", "MyEtherWallet", "Offline", "0xba83e9ce38b10522e3d6061a12
"2017 Jun"  [182674023323763268000, [47, "contribute-status.im", "Fake ICO", "Status", "Offline", "0xadf5b0c2103598fb66a61714152f1d17
"2017 May"  [900000000000000000, [1213, "myetherwallet.io", "Phishing", "MyEtherWallet", "Offline", "0x379ce20c018fb6301c1872c429ec727
"2017 Nov"  [845694082260162337939, [2258, "omisegotoken.com", "Phishing", "OmiseGo", "Offline", "0x7e3d7d8e31b3f472fcd3c1552a9f00913
"2018 Dec"  [632402118500000000, [3222, "centratoken.site", "Phishing", "Centra", "Active", "0x87b4eed5a781d8b847645082410070f002b4
"2018 Mar"  [405922225095042057225, [5820, "muskonline.tech", "Scamming", "Trust-Trading", "Offline", "0xb617b6060e666b03844269a0a0a9
"2018 Oct"  [1362698531731679897684, [5622, "333eth.io", "Scamming", "Ponzi", "Active", "0x311f71389e3de68f7b2097ad02c6ad7b2dde4c71"]
"2018 Sep"  [15345346650989285773245, [5622, "333eth.io", "Scamming", "Ponzi", "Active", "0x311f71389e3de68f7b2097ad02c6ad7b2dde4c71"
"2019 Apr"  [108007925028465243648, [6631, "ico-matic.network", "Phishing", "Matic", "Offline", "0x4256117a02ac880335f8bfbeed63f92ec0
"2019 Feb"  [253847718900000000, [6773, "coinexplorer.tech", "Phishing", "General", "Offline", "1e7to3gbyfep2cpkmhukfkt2jibqx2xaza"
"2019 Jan"  [345347032900000000, [6319, "bitmrtpro.com", "Scamming", "Trust-Trading", "Active", "0x870691b57198f065530e90f8e98c70d0
"2019 Jun"  [302531059800000000, [6785, "58b48d07-110c-487e-8a77-938cfaa8af18.htmlpasta.com", "Scamming", "Trust-Trading", "Active"
"2019 May"  [417224936200000000, [6710, "receive-ethereum.com", "Scamming", "Trust-Trading", "Active", "0x0f746e54ee888e9953ada918d
```

Figure 20: ScamAnalysis_out

Graph:



Graph 3: Changes throughout time

Price Forecasting: Find a dataset online for the price of ethereum from its inception till now. Utilising Spark mllib build a price forecasting model trained on this, the coursework dataset and any other useful information sources you can find. How accurate can you get your forecast within the coursework window to June 2019? How far past June 2019 does your forecast remain accurate? (20-25/50)

Forecasting:

24Open.py

```python
from pyspark.context import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.sql.types import *
from pyspark.sql import functions as f
from pyspark.sql import types as t
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.feature import StandardScaler
from pyspark.ml.regression import LinearRegression
spark = SparkSession.builder.getOrCreate()
#Read files
blocks = spark.read.csv('/data/ethereum/blocks',header=True, inferSchema=True)
contracts = spark.read.csv('/data/ethereum/contracts',header=True, inferSchema=True)
prices = spark.read.csv('/data/ethereum/prices',header=True, inferSchema=True)
contracts = contracts.withColumn('date',contracts['block_timestamp'].cast('date'))
transactions = spark.read.csv('/data/ethereum/transactions', header=True, inferSchema=True)

#Data preprocessing
transactions = transactions.withColumn('date', f.date_format(transactions.block_timestamp.cast(dataType=t.TimestampType()), "yyyy-MM-dd"))
blocks = blocks.withColumn('date', f.date_format(blocks.timestamp.cast(dataType=t.TimestampType()), "yyyy-MM-dd"))
x_difficulty = blocks.groupBy("date").avg("difficulty").withColumnRenamed('avg(difficulty)', 'difficulty')
x_blocksize = blocks.groupBy("date").avg("size").withColumnRenamed('avg(size)', 'blocksize')
x_blockgas = blocks.groupBy("date").sum("gas_used").withColumnRenamed('sum(gas_used)', 'gasused')
x_blocktrnsctn = blocks.groupBy("date").avg("transaction_count").withColumnRenamed('avg(transaction_count)', 'blocktranscount')
x_blocknumber = blocks.groupBy("date").count().withColumnRenamed('count', 'blockcount')
x_trnsctcount = transactions.groupBy("date").count().withColumnRenamed('count', 'transcount')
trnsctvalue = transactions.groupBy("to_address").sum("value").withColumnRenamed('sum(value)', 'transctval')
contractval = contracts.join(trnsctvalue,contracts.address == trnsctvalue.to_address,how="inner")
x_contractval = contractval.groupBy("date").avg("transctval").withColumnRenamed('avg(transctval)', 'contractval')
prices = prices.withColumn('date', f.date_format(prices.Date.cast(dataType=t.TimestampType()), "yyyy-MM-dd"))
x_prices = prices["date","24h Open (USD)"].orderBy("date", ascending=False).withColumnRenamed('24h Open (USD)', 'prvprice')

#Feature extraction
x_feature = x_difficulty.join(x_blocksize,x_difficulty.date == x_blocksize.date,how="inner").drop(x_blocksize.date)
x_feature = x_feature.join(x_blockgas,x_feature.date == x_blockgas.date,how="inner").drop(x_blockgas.date)
x_feature = x_feature.join(x_blocktrnsctn,x_feature.date == x_blocktrnsctn.date,how="inner").drop(x_blocktrnsctn.date)
x_feature = x_feature.join(x_blocknumber,x_feature.date == x_blocknumber.date,how="inner").drop(x_blocknumber.date)
x_feature = x_feature.join(x_trnsctcount,x_feature.date == x_trnsctcount.date,how="inner").drop(x_trnsctcount.date)
```

```python
x_feature = x_feature.join(x_trnsctcount,x_feature.date == x_trnsctcount.date,how="inner").drop(x_trnsctcount.date)
x_feature = x_feature.join(x_contractval,x_feature.date == x_contractval.date,how="inner").drop(x_contractval.date)
x_feature = x_feature.join(x_prices,x_feature.date == x_prices.date,how="inner").drop(x_prices.date)
x_feature = x_feature.withColumn('date', f.date_add(x_feature['date'], 3))

#Define label
y_prices = prices["date","24h Open (USD)"].orderBy("date", ascending=False)
datestotrain = x_feature.select("date")
y_pricestraintest = y_prices.join(datestotrain,y_prices.date == datestotrain.date,how="inner").drop(datestotrain.date)
#Vector including features and labels
xy_featureslabels = y_pricestraintest.join(x_feature,y_pricestraintest.date == x_feature.date,how="inner").drop(x_feature.date)
xy_featureslabels = xy_featureslabels.drop(xy_featureslabels.date)
#Assemble Feature vector, label and normalise
assembler = VectorAssembler(inputCols=["difficulty","blocksize","gasused","blocktranscount","blockcount","transcount","contractval","prvprice"],outputCol="features")
assembeledfeatures = assembler.transform(xy_featureslabels)
scalerfeature = StandardScaler(inputCol="features",outputCol="scaled_features")
scaleddf = scalerfeature.fit(assembeledfeatures)
scaleddata = scaleddf.transform(assembeledfeatures)

#Model training
(train,test) = scaleddata.randomSplit([0.7,0.3])
lr = LinearRegression(featuresCol = 'scaled_features', labelCol='24h Open (USD)', maxIter=10, regParam=0.3, elasticNetParam=0.8)
model = lr.fit(train)

#Model evaluation
evaluation_summary = model.evaluate(test)
print(evaluation_summary.meanAbsoluteError)
print(evaluation_summary.rootMeanSquaredError)
print(evaluation_summary.r2)
predictions = model.transform(test)
predictions.select("prediction","24h Open (USD)","scaled_features").show()
```

Figure 21: 24Open.py

Code Explanation:

- All the required libraries are imported. Here we are making use of **pyspark.sql** library.
- To begin with, all the dataset files transactions, blocks, contracts, prices are read into their respective variables.
- Data is pre-processed and brought into the required format by date.
- Feature extraction is done to consider previous 3 values of 24H Open (USD) field to predict the next value.
- The labels are defined further to normalise the feature vectors and to train the model.
- The data is split into 70/30 training and testing data.
- Linear Regression is used to train the model.
- Lastly, the model is evaluated, and it's mean absolute error, root mean squared error and accuracy are calculated.

Job Execution:

```
bash-4.2$ spark-submit 24Open.py
20/12/13 18:40:59 WARN util.Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 138.37.36.137 instead (on interface eth0)
20/12/13 18:40:59 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
20/12/13 18:41:06 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disabled.
20/12/13 18:41:06 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disabled.
20/12/13 18:42:28 WARN sql.Column: Constructing trivially true equals predicate, 'date#110 = date#110'. Perhaps you need to use aliases.
20/12/13 18:46:39 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeSystemBLAS
20/12/13 18:46:39 WARN netlib.BLAS: Failed to load implementation from: com.github.fommil.netlib.NativeRefBLAS
15.7988364155
29.8227021849
0.985214724816
+-----------------+-------------+-------------------+
|       prediction|24h Open (USD)|    scaled_features|
+-----------------+-------------+-------------------+
|12.950292170460955|     10.723738|[0.04816136558051...|
|14.379931549228134|      11.76883|[0.05364644708195...|
| 14.33490343613225|      12.71811|[0.10978466518714...|
| 96.69569489896529|     89.864235|[0.27694253066093...|
|194.10031443370323| 171.831364664|[1.49141244836012...|
|12.972912231320175|     11.359705|[0.04516507491924...|
| 293.1411632507116|270.7733758237|[2.73641681400933...|
|364.74295403425214|       399.854|[2.50029763849680...|
| 2.258411035363043|       1.95146|[0.00172574628487...|
| 5.023369019848339|      2.363919|[0.00830622718359...|
|11.204920186990421|     12.553567|[0.04619144191049...|
|117.78815215126323|107.9724708075|[1.76485596901714...|
| 709.7197492055775|       684.448|[2.47046748818233...|
|14.893232872902297|     13.943334|[0.03514994915856...|
| 131.7358377682874|115.2563656375|[2.15495513579343...|
|297.98791872604556|       295.775|[2.26863461434268...|
|3.7567329105346055|      0.980235|[0.00686411847326...|
|130.48020122926377|114.1155443288|[2.16415704917392...|
|169.43649247100052| 209.330909643|[2.55825833060935...|
| 301.9456643891387|    292.806751|[2.24403146667248...|
+-----------------+-------------+-------------------+
only showing top 20 rows
```

Figure 22: 24Open_out

- The mean absolute error is 15.79.
- The root mean squared error is 29.822
- The model accuracy is 0.98.

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. (10/50)

```
Gas_1.py
from mrjob.job import MRJob
import time
import re

class Gas(MRJob):

    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

    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, (total, count))

    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, total/count)

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

Figure 23: Gas_1.py

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.

```
"01-2016" 56596270931.31685
"02-2017" 23047230327.254303
"03-2016" 32797039087.356667
"04-2017" 22355124545.395317
"05-2016" 23746277028.263245
"06-2017" 30199442465.128727
"07-2016" 22629542449.24175
"08-2015" 159744029578.03113
"09-2016" 25270403393.626083
"10-2016" 32112869584.914665
"11-2015" 53607614201.796776
"12-2016" 50318068074.68128
"01-2017" 22507570807.719795
"02-2016" 69180681134.38849
"03-2017" 23232253600.81683
"04-2016" 23361180502.721268
"05-2017" 23572314972.01526
"06-2016" 23021251389.812134
"07-2017" 25465699529.05283
"08-2016" 22396836435.95849
"09-2015" 56511301521.033226
"10-2015" 53901692120.53661
"11-2016" 24634294365.279953
"12-2015" 55899526672.35486
```

Figure 24: Gas_1out

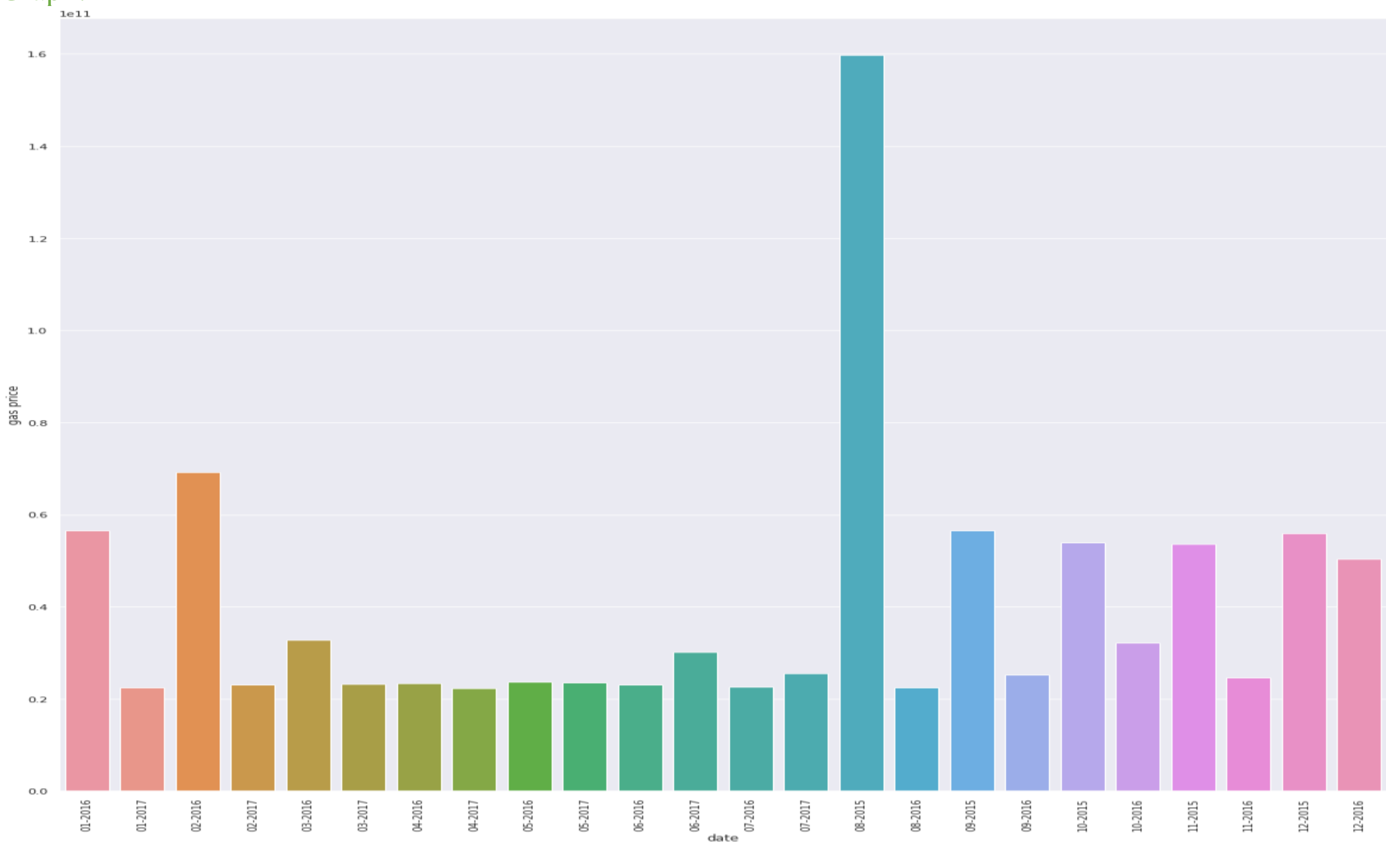
Command Line:

>>python Gas_1.py -r hadoop –output-dir Gas_1_out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/date/ethereum/transactions

Job Id:

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_4356/

Graph:



Graph 4: Gas price change over time

- To calculate average gas price per month.

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

class Gas(MRJob):

    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[4]),1))
        except:
            pass

    def combiner(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, (total, count))

    def reducer(self, feature, values):
        count = 0
        total = 0
        for value in values:
            count += value[1]
            total += value[0]
        yield(feature, total/count)

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

Figure 25: Gas_2.py

Code Explanation:

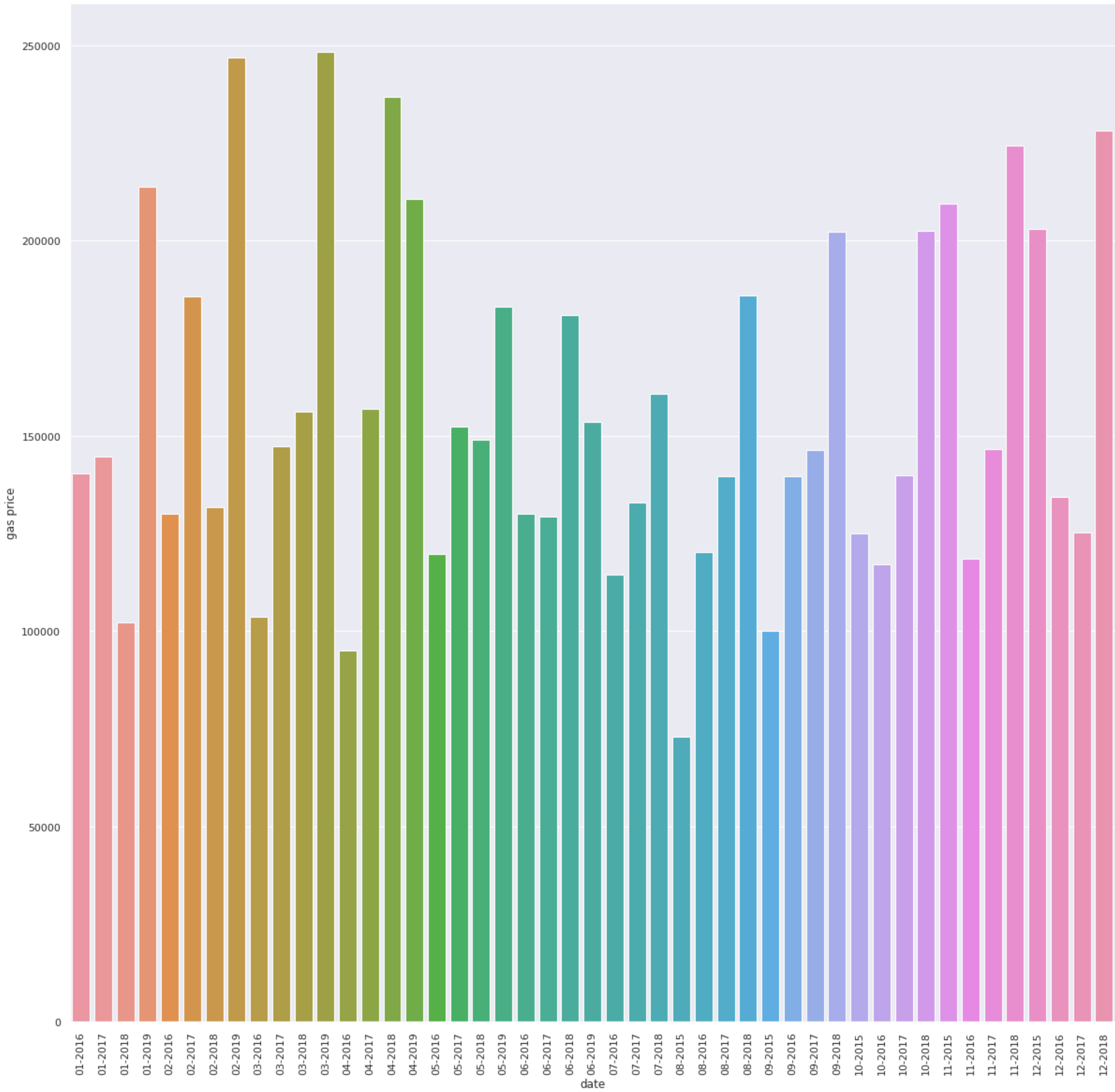
- To analyse the gas price change over years the gas price value in Wei and timestamp is used to perform the time series analysis. The average of gas price value per month gives us an insight of the gas price change over years.
- A map reduce job is used to perform this computation. In mapper **key** 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.

Figure 21: Gasguzzler_out

Command Line:
>>python Gas_1.py -r hadoop –output-dir Gas_2out –no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/date/ethereum/transactions

Job Id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_9061/

Graph:



Graph 5: Average gas price per month

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? (10/50)

- Single spark job to perform all 3 operations from PartB Map/Reduce jobs.
- First step is to calculate the aggregate counts for all transactions in the transactionSmall **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.

PartB_spark.py

```python
import pyspark
import timeit

start = timeit.default_timer()

sc = pyspark.SparkContext()

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

    except:
        return False

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

print(sc._jsc.sc().applicationId())
transactions = sc.textFile('hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactionSmall')
transactions_f = transactions.filter(clean_transactions)
address = transactions_f.map(lambda l: (l.split(',')[2], int(l.split(',')[3]))).persist()
job1output = address.reduceByKey(lambda a,b: (a+b)).sortByKey()
job1output_join = job1output.map(lambda f: (f[0], f[1]))

contracts = sc.textFile('hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts')
contracts_f = contracts.filter(clean_contracts)
contracts_join = contracts_f.map(lambda f: (f.split(',')[0], f.split(',')[3]))
joined_data = job1output_join.join(contracts_join)
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]))

stop = timeit.default_timer()
execution_time = stop - start

print("Program Execution Time: "+str(execution_time))
```

Figure 26: PartB_spark.py

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.

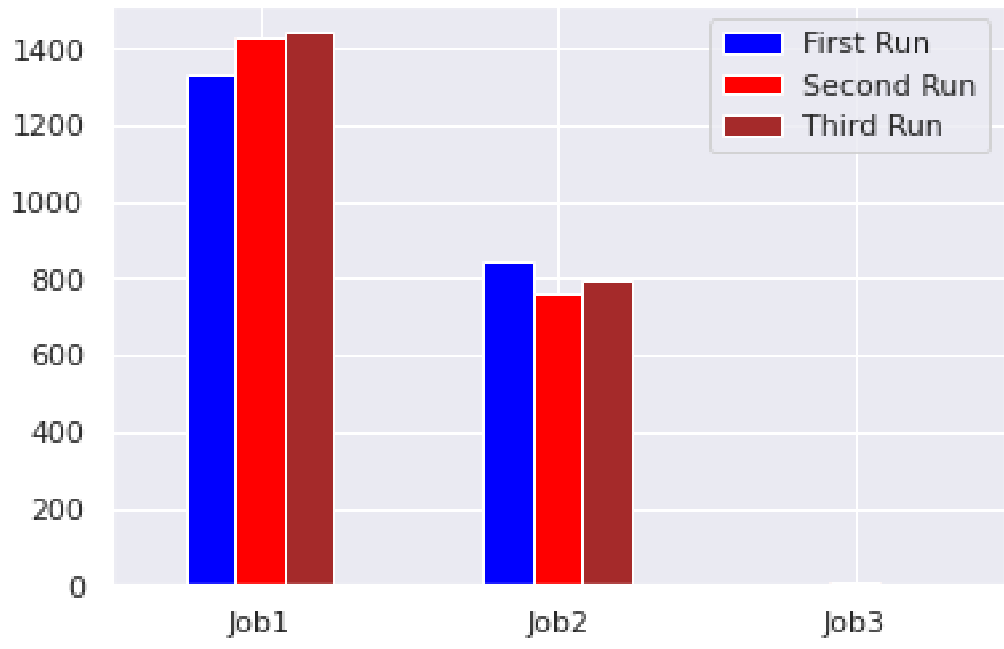
Job Id:
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1607539937312_5763/

Output:

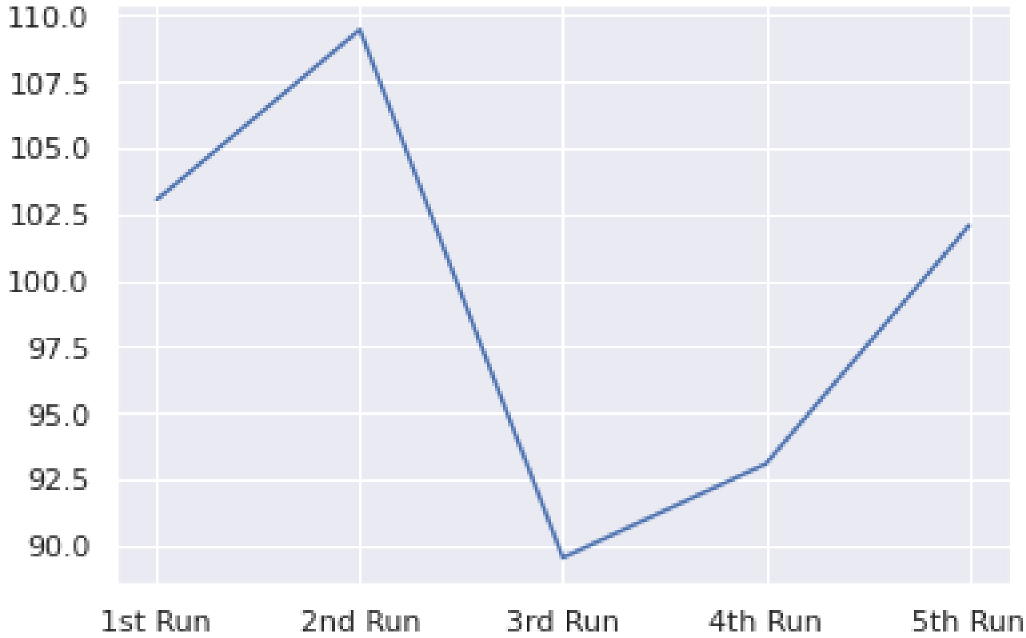
```
bash-4.2$ spark-submit PartB_spark.py
20/12/12 23:13:20 WARN util.Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 138.37.36.239 instead (on interface eth0)
20/12/12 23:13:20 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind to another address
20/12/12 23:13:27 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
20/12/12 23:13:27 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disabled.
application_1607539937312_5763
(1, '0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444, 82572379063979583227674763')
(2, '0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef, 35573259089658511100000000')
(3, '0x7727e5113d1d161373623e5f49fd568b4f543a9e, 30135667743638042022500265')
(4, '0xfa52274dd61e1643d2205169732f29114bc240b3, 24339092574907926349190223')
(5, '0xbfc39b6f805a9e40e77291aff27aee3c96915bdd, 21104195138093660050000000')
(6, '0xbb9bc244d798123fde783fcc1c72d3bb8c189413, 11983593635123926635859368')
(7, '0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8, 8753745137251948111838211')
(8, '0x341e790174e3a4d35b65fdc067b6b5634a61caea, 8379000751917755624057500')
(9, '0xe94b04a0fed112f3664e45adb2b8915693dd5ff3, 5911371784325985597650000')
(10, '0xabbb6bebfa05aa13e908eaa492bd7a8343760477, 5509468260013762691682600')
Program Execution Time: 102.070564032
```

Figure 27: PartB_spark_out

Graph:



Graph 6: Time taken by Hadoop Jobs over multiple runs(s)



Graph 7: Time taken by Spark Job over multiple runs(s)

- 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 maps 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 99.44 seconds and Hadoop jobs is 2207.37 seconds. So it is clear that Spark jobs perform better than Hadoop Map/Reduce jobs.