ETHEREUM ANALYSIS

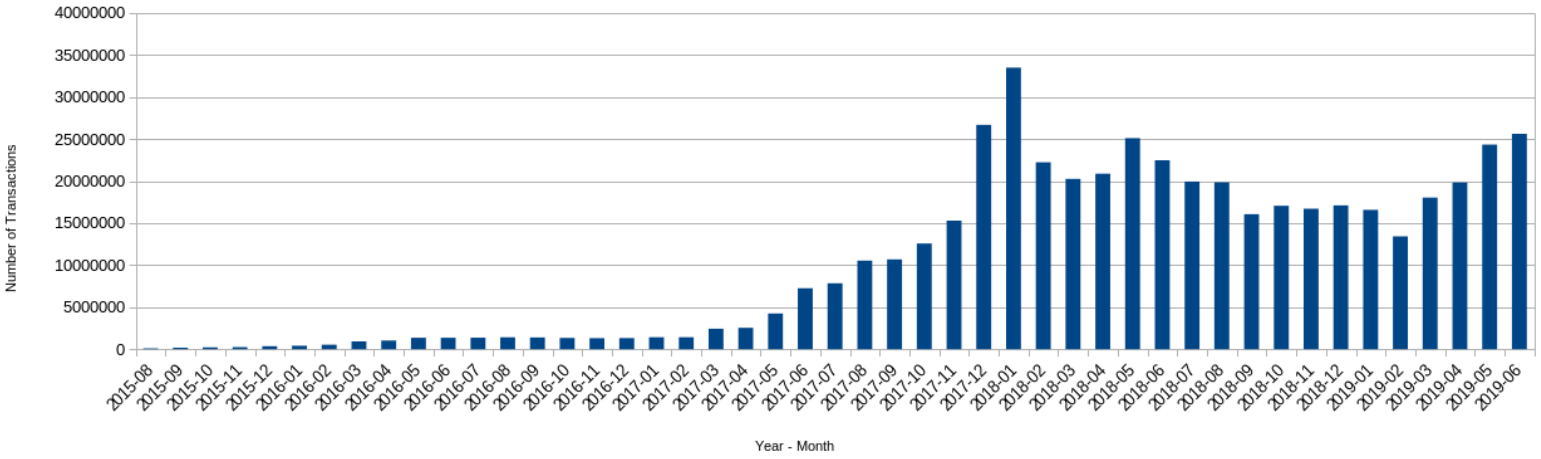
ECS765P

Aris Christofides - 210911099

PART A. TIME ANALYSIS

1. Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.
2. Create a bar plot showing the average value of transactions in each month between the start and end of the dataset.

1.



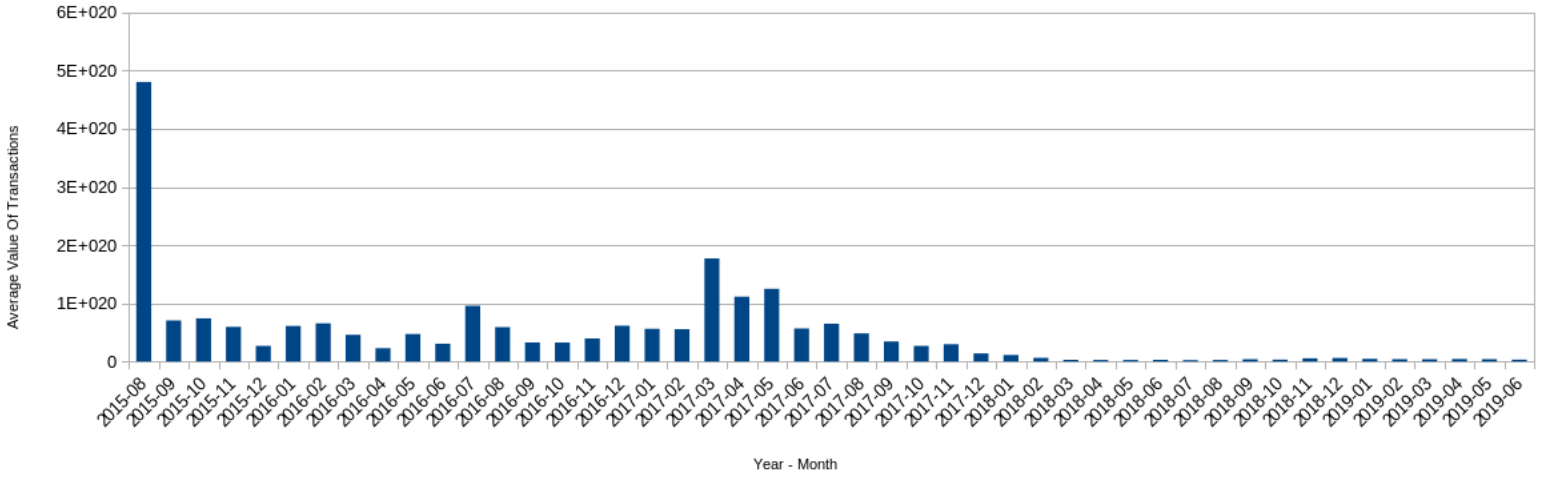
To get the number of transactions occurring every month, I extracted the year and month from the **transactions** dataset in the mapper and then calculated the number of transactions in every month of each year.

Filename: **NumOfTrans.py**

Job ID: **application_1648683650522_0285**

Output Filename: **NumOfTrans.csv**

2.



To get the average value of transactions occurring every month, I extracted the year and month from the **transactions** dataset and the value of each transaction in the mapper. Then I calculated the average value of transactions in every month of each year.

Filename: **AvgValue.py**

Job ID: **application_1648683650522_0328**

Output Filename: **AvgValue.csv**

PART B. TOP TEN MOST POPULAR SERVICES

(Evaluate the top 10 smart contracts by total Ether received)

To get the top ten most popular services I used the following procedure:

The first mapper receives lines from both **transactions** and **contracts** datasets. I then check where each line came from by checking the length of the returned array when the line has been split. The lines from the **transactions** dataset have 7 fields, while the lines from the **contracts** dataset have 5 fields. Once the line has been distinguished, I extracted out the information needed and yielded it to the reducer. The key for the mapper's output is the address, which allows the join to take place on the reducer. The value contains a field to indicate the source of the information. So "value" is for **transactions**, and "contract" for the **contracts**. In the first reducer, I created a loop around all values yielded for a given key. In each loop, I use the first field in the value to distinguish the information included, adding the value from **transactions** and filtering the addresses that were not present within **contracts**. This reducer yields contract addresses and the sum value of each address. The second MRJob sorts out pairs from the first one and yields the first ten pairs.

Filename: **TopTenServices.py**

Job 1 ID: **application_1648683650522_0355**

Job 2 ID: **application_1648683650522_0389**

Output:

"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	**84155100809965865822726776**
"0xfa52274dd61e1643d2205169732f29114bc240b3"	**45787484483189352986478805**
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"	**45620624001350712557268573**
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"	**43170356092262468919298969**
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"	**27068921582019542499882877**
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"	**21104195138093660050000000**
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"	**15562398956802112254719409**
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"	**11983608729202893846818681**
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"	**11706457177940895521770404**
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"	**83790007519177556240575 00**

Output Filename: **TopTenServicesOutput.txt**

PART C. TOP TEN MOST ACTIVE MINERS

(Evaluate the top 10 miners by the size of the blocks mined)

To get the top ten most active miners, the mapper extracts data from the **blocks** dataset. And the first reducer aggregates the block size for each miner. Then, the second reducer sorts the result and yields the top ten miners.

Filename: **TopTenMiners.py**

Job 1 ID: **application_1648683650522_4568**

Job 2 ID: **application_1648683650522_4571**

Output:

"0xea674fdde714fd979de3edf0f56aa9716b898ec8"	**23989401188**
"0x829bd824b016326a401d083b33d092293333a830"	**15010222714**
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c"	**13978859941**
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"	**10998145387**
"0xb2930b35844a230f00e51431acae96fe543a0347"	**7842595276**
"0x2a65aca4d5fc5b5c859090a6c34d164135398226"	**3628875680**
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"	**1221833144**
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"	**1152472379**
"0x1e9939daaad6924ad004c2560e90804164900341"	**1080301927**
"0x61c808d82a3ac53231750dadc13c777b59310bd9"	**692942577**

Output Filename: **TopTenMinersOutput.txt**

PART D. DATA EXPLORATION

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?)

For this task, I reimplemented Part B in Spark and I obtained the same results with the MapReduce version:

(u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', 84155100809965865822726776L)

(u'0xfa52274dd61e1643d2205169732f29114bc240b3', 45787484483189352986478805L)

(u'0x7727e5113d1d161373623e5f49fd568b4f543a9e', 45620624001350712557268573L)

(u'0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', 43170356092262468919298969L)

(u'0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8', 27068921582019542499882877L)

(u'0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', 21104195138093660050000000L)

(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', 15562398956802112254719409L)

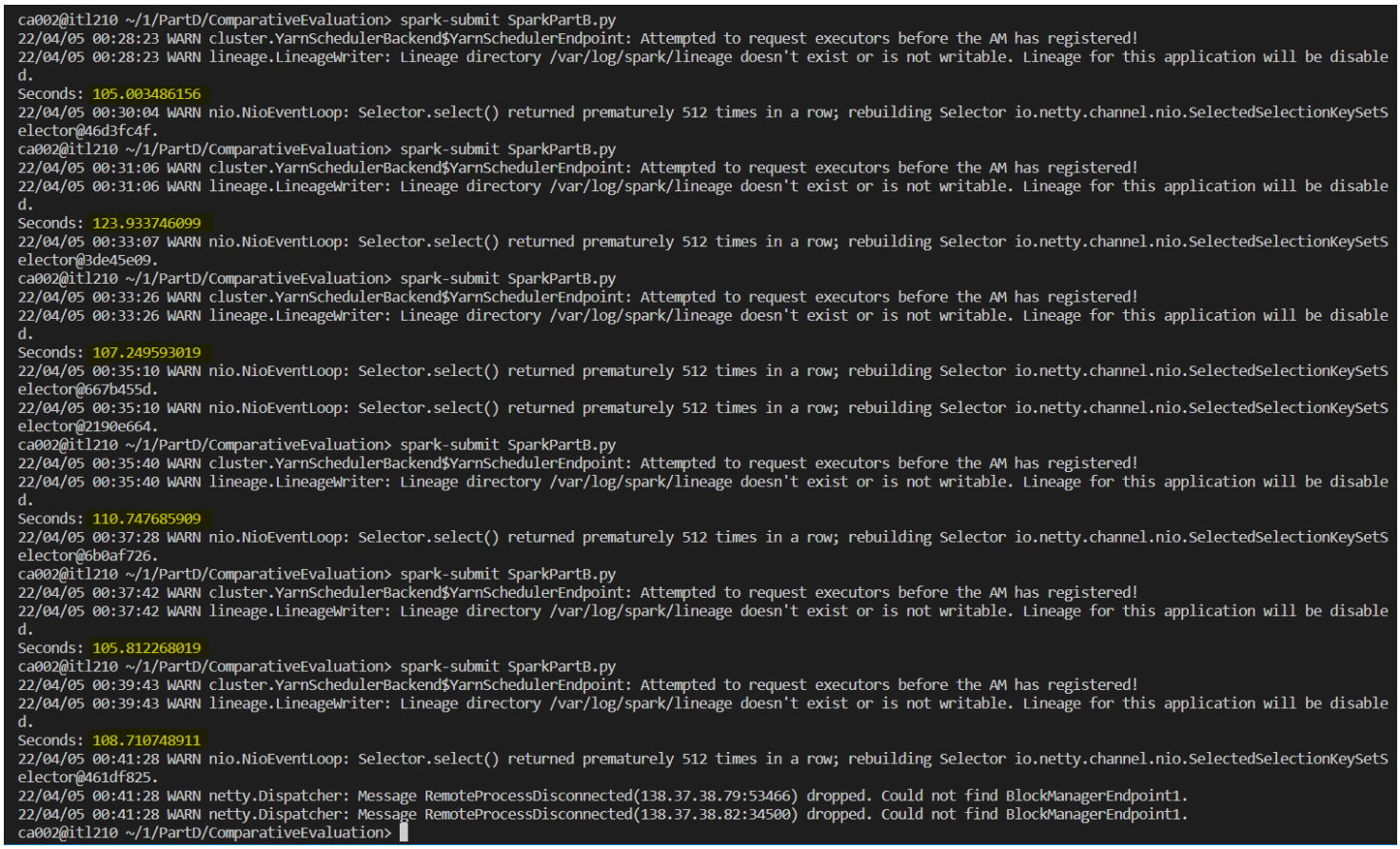
(u'0xbb9bc244d798123fde783fcc1c72d3bb8c189413', 11983608729202893846818681L)

(u'0xabbb6bebfa05aa13e908eaa492bd7a8343760477', 11706457177940895521770404L)

(u'0x341e790174e3a4d35b65fdc067b6b5634a61caea', 83790007519177556240575OOL)

Output Filename: **SparkPartBOutput.txt**

I commented out the print function and executed the job 6 times in order to calculate the average performing time:



```
ca002@itl210 ~/1/PartD/ComparativeEvaluation> spark-submit SparkPartB.py
22/04/05 00:28:23 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
22/04/05 00:28:23 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disable
d.
Seconds: 105.003486156
22/04/05 00:30:04 WARN nio.NioEventLoop: Selector.select() returned prematurely 512 times in a row; rebuilding Selector io.netty.channel.nio.SelectedSelectionKeySetS
elector@46d3fc4f.
ca002@itl210 ~/1/PartD/ComparativeEvaluation> spark-submit SparkPartB.py
22/04/05 00:31:06 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
22/04/05 00:31:06 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disable
d.
Seconds: 123.933746099
22/04/05 00:33:07 WARN nio.NioEventLoop: Selector.select() returned prematurely 512 times in a row; rebuilding Selector io.netty.channel.nio.SelectedSelectionKeySetS
elector@3de45e09.
ca002@itl210 ~/1/PartD/ComparativeEvaluation> spark-submit SparkPartB.py
22/04/05 00:33:26 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
22/04/05 00:33:26 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disable
d.
Seconds: 107.249593019
22/04/05 00:35:10 WARN nio.NioEventLoop: Selector.select() returned prematurely 512 times in a row; rebuilding Selector io.netty.channel.nio.SelectedSelectionKeySetS
elector@667b455d.
22/04/05 00:35:10 WARN nio.NioEventLoop: Selector.select() returned prematurely 512 times in a row; rebuilding Selector io.netty.channel.nio.SelectedSelectionKeySetS
elector@2190e664.
ca002@itl210 ~/1/PartD/ComparativeEvaluation> spark-submit SparkPartB.py
22/04/05 00:35:40 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
22/04/05 00:35:40 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disable
d.
Seconds: 110.747685909
22/04/05 00:37:28 WARN nio.NioEventLoop: Selector.select() returned prematurely 512 times in a row; rebuilding Selector io.netty.channel.nio.SelectedSelectionKeySetS
elector@6b0af726.
ca002@itl210 ~/1/PartD/ComparativeEvaluation> spark-submit SparkPartB.py
22/04/05 00:37:42 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
22/04/05 00:37:42 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disable
d.
Seconds: 105.812268019
ca002@itl210 ~/1/PartD/ComparativeEvaluation> spark-submit SparkPartB.py
22/04/05 00:39:43 WARN cluster.YarnSchedulerBackend$YarnSchedulerEndpoint: Attempted to request executors before the AM has registered!
22/04/05 00:39:43 WARN lineage.LineageWriter: Lineage directory /var/log/spark/lineage doesn't exist or is not writable. Lineage for this application will be disable
d.
Seconds: 108.710748911
22/04/05 00:41:28 WARN nio.NioEventLoop: Selector.select() returned prematurely 512 times in a row; rebuilding Selector io.netty.channel.nio.SelectedSelectionKeySetS
elector@461df825.
22/04/05 00:41:28 WARN netty.Dispatcher: Message RemoteProcessDisconnected(138.37.38.79:53466) dropped. Could not find BlockManagerEndpoint1.
22/04/05 00:41:28 WARN netty.Dispatcher: Message RemoteProcessDisconnected(138.37.38.82:34500) dropped. Could not find BlockManagerEndpoint1.
ca002@itl210 ~/1/PartD/ComparativeEvaluation>
```

(105+123.93+107.24+110.74+105.81+108.71) / 6 = 110.24

Average performing time is 110.24 seconds. MapReduce takes around 30-40 minutes to perform the same task. Surely, Spark is the better framework for this task. The reason is because Spark performs in-memory processing to avoid unnecessary Input/Output operations.

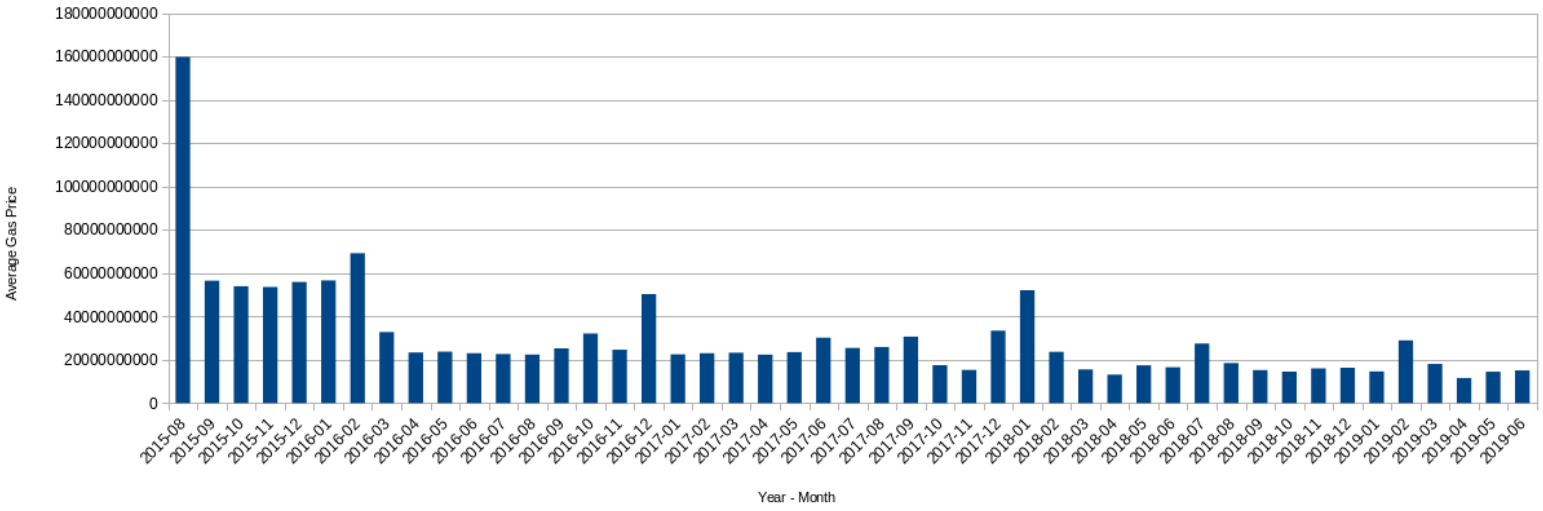
Filename: **SparkPartB.py**

Job ID: **application_1648683650522_2336**

Gas Guzzlers:

(How has gas price changed over time? Have contracts become more complicated, requiring more gas, or less so? Also, could you correlate the complexity for some of the top-10 contracts found in Part-B by observing the change over their transactions)

To get the price change over time, the mapper extracts the month and gas price from the **transactions** dataset. And then, the reducer calculates the average value of each month.



As we can see from the chart, initially the average gas price was extremely high, but in the next month it dropped approximately 60%.

Filename: **GasPriceChange.py**

Job ID: **application_1648683650522_5796**

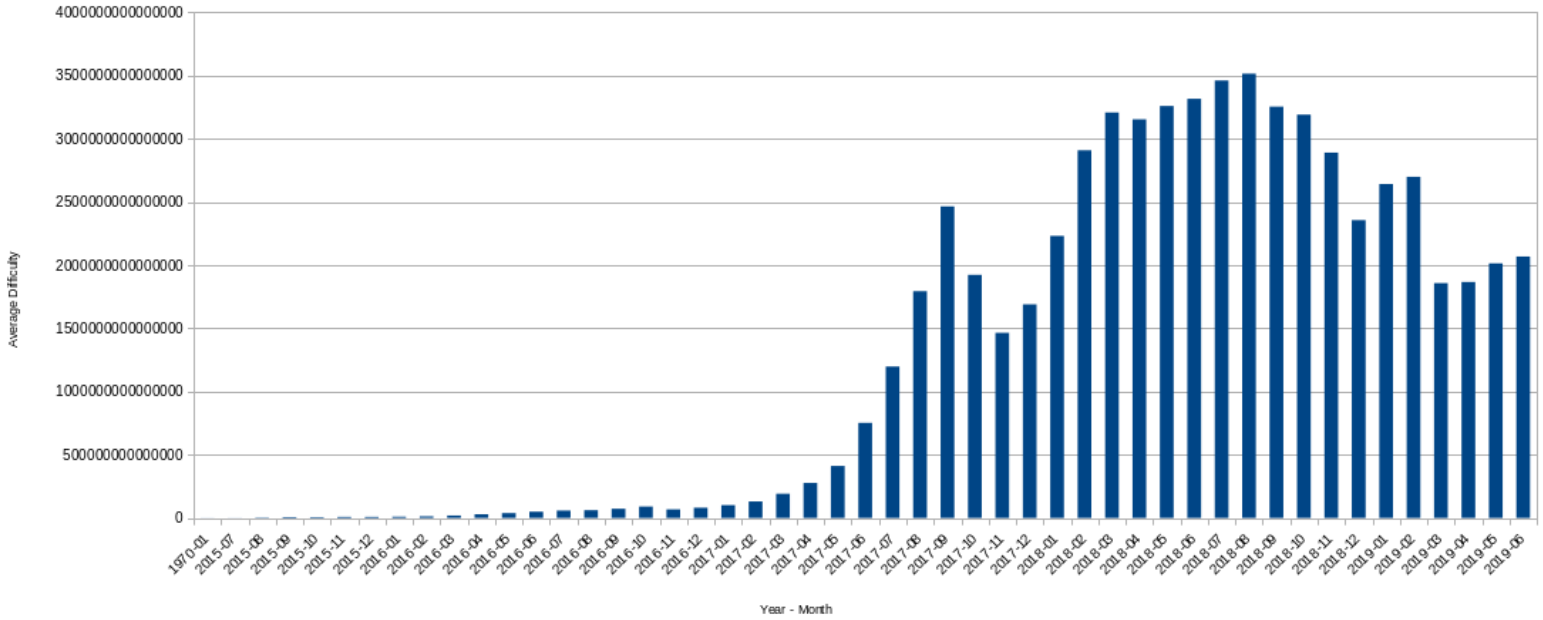
Output Filename: **GasPriceChange.csv**

To understand the relationship between contract complexity and gas required, I plotted 2 graphs. For the first one I used the difficulty and timestamp from the **blocks** dataset. The mapper extracts the difficulty and year-month from **blocks**, and the reducer calculates the average difficulty for each month. For the second graph I extracted the gas used and the timestamp from **blocks** and calculated the average for gas used over time.

Filename: **ContractComplexity.py** Filename: **ContractComplexity2.py**

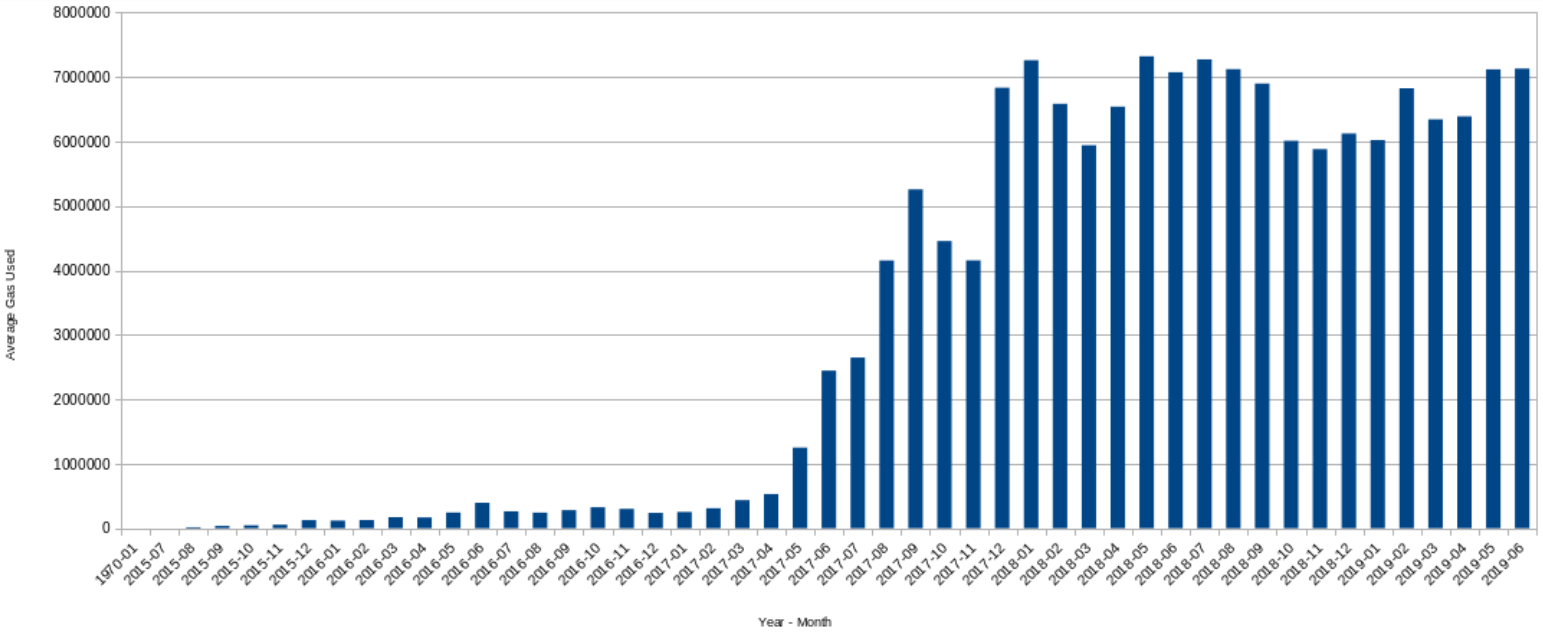
ContractComplexity Job ID: **application_1648683650522_5947**

ContractComplexity2 Job ID: **application_1649894236110_1795**



The graph shows the average difficulty over time. As it seems, the difficulty over time went up, whereas gas price went down

Output Filename: **ContractComplexity.csv**



This graph shows the average gas used over time. It is clear by looking at both graphs, that difficulty and gas used are positively correlated. The more difficult the transaction is, the more gas is used.

Output Filename: **ContractComplexity2.csv**

To correlate this with the results from part B, I used the top ten addresses from Part B to filter the address and block number from the **transactions** dataset, and extracted block number, difficulty and gas used from the **blocks** dataset. In the mapper, I used the block number as the key to join the addresses, the difficulty and the gas used for transactions in the address.

Filename: **TopTenServicesCorrelation.py**

Job ID: **application_1648683650522_6017**

Output:

"1946708" **["0xe94b04a0fed112f3664e45adb2b8915693dd5ff3", 58077875732620, 575668]**

"1966054" **["0xfa52274dd61e1643d2205169732f29114bc240b3", 54442807226771, 375125]**

"1969372" **["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", 56794368459018, 1101578]**

"2462919" **["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", 92108407601593, 464781]**

"1428757" **["0xbb9bc244d798123fde783fcc1c72d3bb8c189413", 32880398612201, 3711215]**

"1919996" **["0x341e790174e3a4d35b65fdc067b6b5634a61caea", 62443940864281, 734742]**

"1920419" **["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", 58309072372499, 731981]**

"1935363" **["0xbfc39b6f805a9e40e77291aff27aee3c96915bdd", 59769221734766, 228056]**

"2041128" **["0x7727e5113d1d161373623e5f49fd568b4f543a9e", 56932397526793, 139259]**

"2206259" **["0xabbb6bebfa05aa13e908eaa492bd7a8343760477", 68423714256992, 1353738]**

Output Filename: **TopTenServicesCorrelation.txt**

The first column is the block number, the second is the address, the third is the difficulty and the fourth is the gas used.

It seems that the top ten addresses have a large number of difficulty but relatively low number in gas used.

Scam Analysis:

(What is the most lucrative form of scam? Produce a table with SCAM TYPE, STATE, VOLUME)

To be able to perform the scam analysis I firstly had to download the **scams.json** dataset from the cluster. Then I converted the **scams.json** dataset to csv in order to get 3 Columns (Address, Type of Scam, Status), to be able to perform the analysis.

Filename: **conversion.py**

Then I created a spark file to perform both tasks. I used the **transactions** dataset to get the "to address" and value, and created a dataframe that reads from the created **scams.csv** file.

Filename: **ScamAnalysis.py**

Job ID: **application_1649894236110_1867**

The first job extracts the scam type, gas used and total wei and reduces to get the most lucrative scams.

Output:

(u'Phishing', (724922480.0, 4.3727010025040765e+22))

(u'Fake ICO', (7402302.0, 1.3564575668896297e+21))

(u'Scamming', (5978937963.0, 4.471580609844053e+22))

Output Filename: **MostLucrativeScams.txt**

The second job gets the scam type, status, gas used and total wei.

Output:

((u'Fake ICO', u'Offline'), (7402302.0, 1.3564575668896297e+21))

((u'Phishing', u'Active'), (111750772.0, 6.256455846464806e+21))

((u'Phishing', u'Inactive'), (2255249.0, 1.488677770799503e+19))

((u'Phishing', u'Offline'), (610020459.0, 3.745402749273797e+22))

((u'Phishing', u'Suspended'), (896000.0, 1.63990813e+18))

((u'Scamming', u'Active'), (4282186989.0, 2.2612205279194265e+22))

((u'Scamming', u'Offline'), (1694248234.0, 2.209989065129632e+22))

((u'Scamming', u'Suspended'), (2502740.0, 3.71016795e+18))

Output Filename: **ScamsTable.txt**

Wash Trading:

(Which addresses are involved in wash trading? Which trader has the highest volume of wash trades?)

To obtain the wash trading addresses I used the “to” and “from addresses” from the **transactions** dataset as join keys, to see where there is some sort of cycle of value. This filtering stage occurs in the mapper and in the reducer.

Filename: **WashTradingAddresses.py**

Job ID: **application_1649894236110_3638**

Some of the Wash Trading Addresses: (All can be found in the output txt file)

Output:

"0x00000614cd7682f17f963163ecbba11dead32061"	**90**
"0x00000b3c3794503a8c664439d14e01c4c80b750c"	**669**
"0x000042fca0f044aeca7f7b3903d933bba521b240"	**76**
"0x00005b5cfe44946fe8218feb315c12362d3ec3d5"	**71**
"0x000074887dfa452b0ea898365e9977debc3206e2"	**11**
"0x00009fd1e72f1a635a13a8e4177c697886033ba7"	**13**
"0x0000b446bc8b47be5ef8d5ea8964508130e77ce0"	**80**
"0x0000c2dfc27ba3be1369a09d849cb63843070519"	**123**
"0x00013237369447ba252e1ec486e8309071470d01"	**11**
"0x000187141609d9a744a34eaf9e716a97a8a5cd00"	**17**
"0x00018ee0618aae5e04b6cf3f31a404aea48f82dc"	**11**
"0x0001a14487834de52ead6872b8ba233f943c01fc"	**11**
"0x00020b602ad4b7e8283a53300a932de3a2ab805e"	**13**
"0x00034ab4c14325576bea80274e17e21a17ba73a0"	**21**
"0x0003ad9b8c1699106a222f22504cda086750b9cb"	**12**

Output Filename: **WashTradingAddresses.txt**

To get the trader with the highest volume of wash trades I used the same code but added a reducer to get the top ten values.

Filename: **TopWashTradingAddresses.py**

Job 1 ID: **application_1649894236110_3696**

Job 2 ID: **application_1649894236110_3739**

Output:

1 "0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be,852231"

2 "0x876eabf441b2ee5b5b0554fd502a8e0600950cfa,274226"

3 "0x0d0707963952f2fba59dd06f2b425ace40b492fe,245041"

4 "0x5e032243d507c743b061ef021e2ec7fcc6d3ab89,243312"

5 "0x7ed1e469fcb3ee19c0366d829e291451be638e59,239634"

6 "0x390de26d772d2e2005c6d1d24afc902bae37a4bb,229560"

7 "0x2b5634c42055806a59e9107ed44d43c426e58258,206161"

8 "0x6cc5f688a315f3dc28a7781717a9a798a59fda7b,176563"

9 "0x75e7f640bf6968b6f32c47a3cd82c3c2c9dcae68,165805"

10 "0x69ea6b31ef305d6b99bb2d4c9d99456fa108b02a,123396"

Output Filename: **TopWashTradingAddresses.txt**

Number 1, **0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be** has the highest volume of wash trades with **852231** trades**.**