Coursework

Part A

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

For this task I utilised the MRjob python framework to work on the Transactions dataset. This was done by doing a count job with the transaction date (month and year) being the key.

To do this I ran the command

```
python A1.py -r hadoop --output-dir A1OUT --no-cat-output
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions
```

This produced a result where the result was a date and the total transaction associated with said date, which can be found with this document (A1OUT.tsv). More explaination of the code can be found in A1.py file.

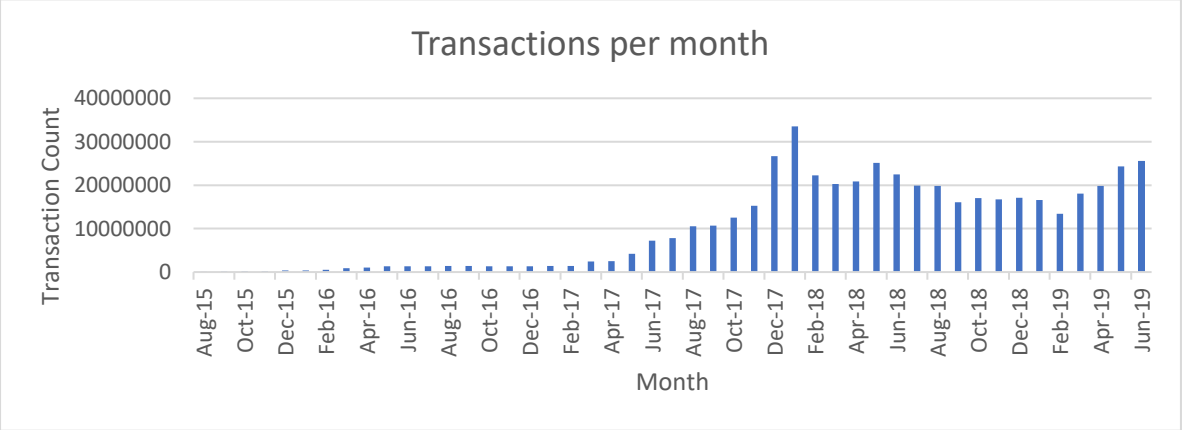


Figure 1 - number of transactions a month on the Ethereum blockchain

This graph shows there was a spike in transactions in January of 2018, which may correspond to an event and some excitement which caused Ethereum users to utilise the cryptocurrency more than they usually do.

The average transactions job was like total transactions (the previous job), this is because to calculate average count must be utilised.

I ran the following command to achieve this:

```
python A2.py -r hadoop --output-dir A2OUT --no-cat-output
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions
```

To do this I first mapped the transaction value instead of 1 (which is required for count jobs). This means the reducer would receive the generator of all transactions that month. After this, I would iterate through this generator and calculate the total as well as the count of items (as average is total/count), then these totals would be generated. More explaination of the code can be found in A2.py file. The output can be found with this document (A2OUT.tsv)

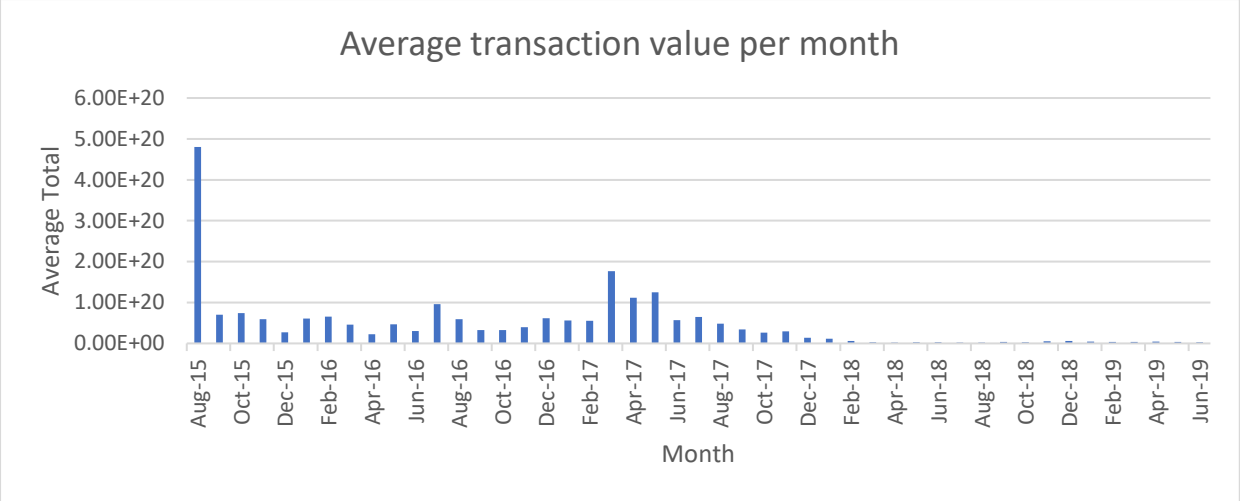


Figure 2 - average value of transactions a month on the Ethereum blockchain (measured in Wei)

This increased number of transactions in the month of August 2015 (as shown by figure 2) is most likely because of the release date being at that time so this caused excitement and intrigue about the cryptocurrency.

Part B

With part B, I split the 3 jobs into 3 sperate jobs (without using MRStep), however the third job was just an extension of the second (adding a top ten filter to the total contact values), so I elected to use MRStep with these 2 parts.

With this part, I first considered repetition join but after doing the first job to get the total transactions for each recipient the resulting file was over 3.5gb. To avoid a potential error 240 (out of space on the HDFS), the best way to operate on 2 files of this magnitude would be to use the alternative join: repartition. This would require inputting both files as streams into the MapReduce job. This can be done in 2 ways:

1. ```
   python B2.py -r hadoop --output-dir B2_out --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts hdfs://andromeda.eecs.qmul.ac.uk/user/qbo30/B1_out.tsv/
   ```

2. ```
   python B2.py B1_out.tsv -r hadoop --output-dir B2_out --no-cat-output hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts
   ```

The extended comments on the code can be found in B1.py, B2.py and B3.py. This goes in depth about the way the task has been executed.

The following is the result of all 3 parts being run

This result is structured like so:

rank **address**, **aggregate amount**

1 "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,841551008099658658227267776"

2 "0xfa52274dd61e1643d2205169732f29114bc240b3,45787484483189352986478805"

3 "0x7727e5113d1d161373623e5f49fd568b4f543a9e,456206240013507125572685 73"

4 "0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,43170356092262468919298969"

5 "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,27068921582019542499882877"

6 "0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,21104195138093660050000000"

7 "0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,155623989568021122547194 09"

8 "0xbb9bc244d798123fde783fcc1c72d3bb8c189413,11983608729202893846818681"

9 "0xabbb6bebfa05aa13e908eaa492bd7a8343760477,11706457177940895521770404"

10 "0x341e790174e3a4d35b65fdc067b6b5634a61caea,8379000751917755624057500"

From this output, we can conclude that the contract with address 0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 has gotten the most transitions to it. This contract is called “Replay Safe Split” and is extended by many other smart contracts. The second highest belongs to the exchange kraken, and the third is another exchange: Bitfinex.

Part C

In part C I elected to use spark instead of map reduce. This was a simple aggregate job, with the goal being to add up the size in the miner field for blocks and then calling a spark top ten method to sort to the miners in terms of aggregate size.

This was run using

```
spark-submit C.py > COUT.tsv
```

The job was essentially just working with blocks to get the top 10 miners by size, more of the explanation step by step can be found in C.py

The result is as follows (found in COUT.tsv)

This result is structured like so:

rank **miner address**, **aggregate size**

1. 0xea674fdde714fd979de3edf0f56aa9716b898ec8: 23989401188

2. 0x829bd824b016326a401d083b33d092293333a830: 15010222714

3. 0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c: 13978859941

4. 0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5: 10998145387

5. 0xb2930b35844a230f00e51431acae96fe543a0347: 7842595276

6. 0x2a65aca4d5fc5b5c859090a6c34d164135398226: 3628875680

7. 0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01: 1221833144

8. 0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb: 1152472379

9. 0x1e9939daaad6924ad004c2560e90804164900341: 1080301927

10. 0x61c808d82a3ac53231750dadc13c777b59310bd9: 692942577

From the operation we can conclude that the miner 0xea674fdde714fd979de3edf0f56aa9716b898ec8 is the largest.

Part D

Scam Analysis

1. Popular Scams

For this task I chose the analyse the top 10 most lucrative scams (the most popular scams) as opposed to the scam categories, this analysis was based on the growth of the aggregate amount being sent to the scam addresses associated with those scams IDs over time, then I would find the last know date that each offline scam was associated with, and with those offline dates I would isolate the fastest growing scam and see if they correspond to any other scams going offline . This was done through the following series of steps:

a. Converting the scams json to a csv
b. Joining the scams json to transactions to get which scam id got the most ETH transferred to it
    i. Yielding every address associated with scam
    ii. Yielding every to address of the transactions
    iii. Finding matches of addresses
    iv. Combining all the aggregates of these addresses to the single scam id
    v. Yielding each scam id with their totals
    vi. Ranking these aggregate scam totals
c. Using the top 10 scam ids and addresses to get information on how these scams have changed throughout time.
    i. This task is like the one above except utilising the transaction dates themselves in the reduces
d. Finding offline dates of scams
    i. Finding all offline scams using the scams Jason to csv converter
    ii. Getting every date associated with that scam
    iii. Finding the largest date associated with that scam and yielding that instead of the entire time series

The precise steps to achieve this via map reduce jobs can be found in the files:

top10Scams.py, top10throughTime.py, offlineDates.py

Some extra computation has been done in terms of changing the json to csv code and for plotting figure 3

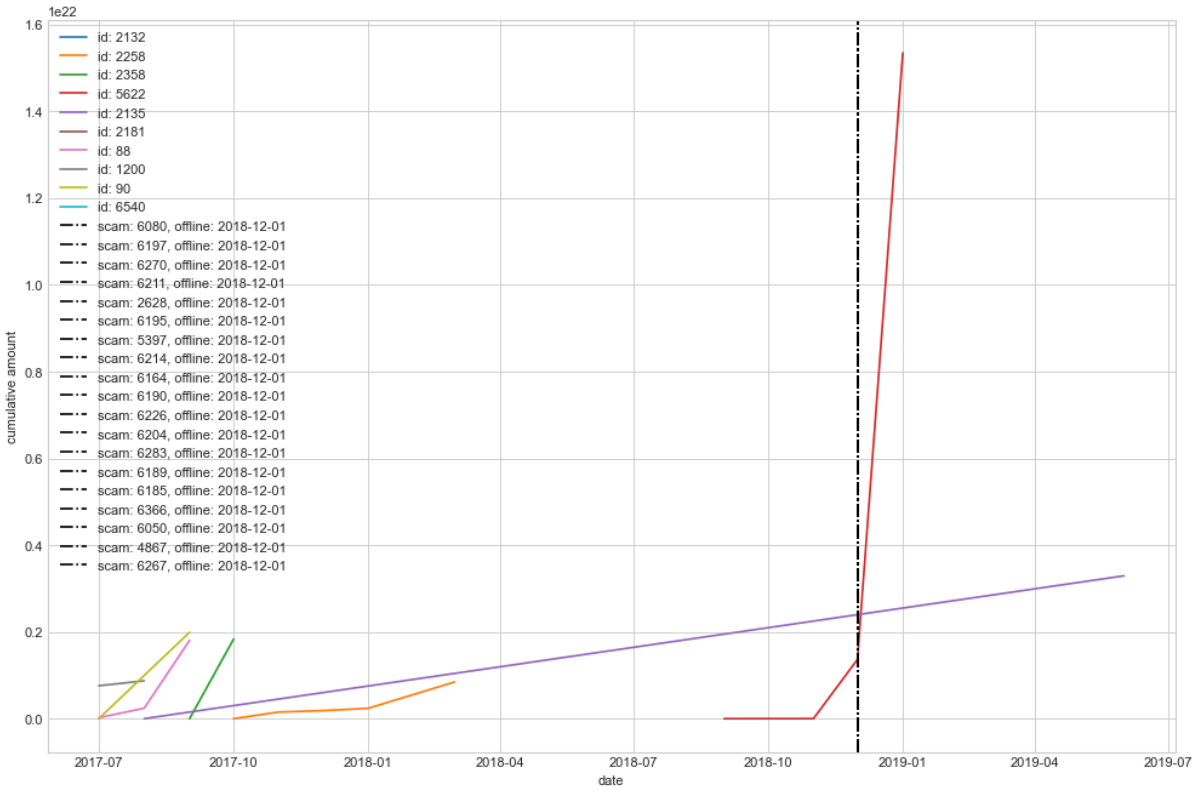


Figure 3 - top 10 ETH scams over time

each file is run with:

jsonToCSV.py –

```
hadoop fs -get /data/ethereum/scams.json
```

```
python jsonToCSV.py > scams.csv
```

top10scams.py –

```
python top10scams.py scams.csv -r Hadoop
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions >
top10scams.tsv
```

top10throughTime.py –

```
python top10throughTime.py scams.csv -r Hadoop
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions –file
top10scams.tsv > ScamsOverTimeOut.tsv
```

offliineDates.py –

```
python top10throughTime.py offline.csv -r Hadoop
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions >
offlineDates.tsv
```

Each of the top 10 scams have their own unique patterns of aggregate growth (however 3 have not made enough compared to the others to fully appear in figure 1), this is likely because of the different types of scams that they are and therefore the different strategies that they use to get investors to send ETH to their addresses.

The fastest spiking scam (5622), with the name 333eth.io is a Ponzi scheme which promised the daily return of 3.33% of their investment back (cryptoinfowatch, 2021). This was at one point the most popular decentralised app on the Ethereum blockchain (allegedly), as seen by the graph: it saw an almost exponential increase of revenue after launch, with the last date in the dataset associated with 333eth earning it as cumulative revenue of over 1.4e22 Wei which converts to roughly 46,215,878.3039 GBP. While the data says the scam is still active, its website has not been renewed (however it's twitter is still available but is not active). This project yielded such a high reward for the team because of the excitement generated by the ability to get ETH by simply investing in a service without trading, however in reality: the owners were just circulating the ETH received from other users to make it appear as though the service was legitimate. The downfall of 333Eth likely came when it was flagged as a scam by Metamask ("a popular browser extension to run Ethereum based dApps in your browser" (Meears, 2018))

In terms of scams that have gone offline in the time around 333Eth's rise in income, the majority of these have been Trust Trading scams (ids: 6080, 6197, 6270, 6211, 6195, 6214 etc.). The rapid offline nature of these trust trading scams is most likely down to the simplicity of many of them. Having the user send ETH to the scammer in some way for some sort of return or cause. 333Eth is a much more sophisticated scam than that, being a Ponzi scheme, which can run on until there is not enough money to keep up the façade.

The longest running of the top 10 scams: id 2135, this scam is a OmiseGO smart contract impersonator "OmiseGO.com". With OmiseGO being a "Layer 2 Optimistic Rollup scaling solution" (OmiseGO, 2021) smart contract on the Ethereum network which aims to "improve gas fees and extend capabilities of smart contracts" (OmiseGO, 2021) on the Ethereum network. OmiseGO.com's associated website is not currently up but it appears that it has managed to gain consistent revenue because of its unassuming resemblance to the legitimate website. This is much more lucrative in the long run than a Ponzi scheme like 333Eth because it is less in the public eye, but it also has less of a potential to make as much as 333Eth because of that same reason.

The other scams: Ziber.io (2132), omisegotoken.com (2258), worldofbattles.io (2358), atlant.solutions (2181), myetherwallet.com.cm (88), myetherwallet.in (1200), myetherwallet.tech (90), usddex.io (6540) were not as lucrative or as long lived as OmiseGO.com or 333Eth. While some other scams like omisegotoken.com (2258) have also tried to phish using OmiseGO's name, OmiseGO.com is likely the most lucrative because of the ease at which someone could type that website as opposed to the legitimate omg.network. Another very lmucrative form of scam is imitating myetherwallet, a very popular wallet solution for Ethereum. As said before, phishing is not that lucrative in the long run unless you manage to capture the website which most people would expect the project to be hosted on (myetherwallet is hosted on myetherwallet.com as opposed to OmiseGO's omg.network).

2. **Wash Trading**

   Wash trading is falsifying transactions to make it appear that there is more volume than there is, which fuels speculation and makes investors incorrectly believe in a potential feature for the company or currency. This Is fraudulent and is viewed as illegal in the stock market and is the same in the crypto market.

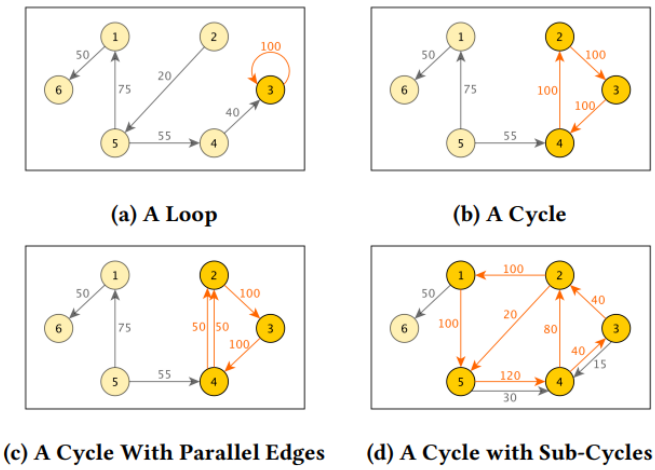


Figure 4- Four examples of directed token trade multigraphs.

Wash trading involves the return equivalent value back to an original node. This means in order to find wash trades within the transactions, these trades would have to fit 3 criteria:

1. The trades would have to return to a single node
2. The value of the transactions must be similar or equal
3. The trades must have occurred multiple times to make a difference on the trade volume

From these criteria, we can then get the addresses that may potentially be the best fits for wash trading.

This can be done by using 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, we can utilise the criteria we've established to get a concrete picture as to which certain addresses have been involved in wash trading.

This job is run using the following commands:

washtradingAddr.py –

```
python washtradingAddr.py -r Hadoop
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions > WT.tsv
```

washtradingAddrTop10.py –

```
python washtradingAddrTop10.py -r Hadoop
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions > top10WT.tsv
```

In the code we have isolated start addresses based upon the criteria and got the following top 10 addresses (fount in top10WT.tsv) with the number of cycles associated with them. This means these addresses are most likely engaging in wash trading.

1 "0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be,852238"

2 "0x876eabf441b2ee5b5b0554fd502a8e0600950cfa,274208"

3 "0x0d0707963952f2fba59dd06f2b425ace40b492fe,245033"

4 "0x5e032243d507c743b061ef021e2ec7fcc6d3ab89,243306"

5 "0x7ed1e469fcb3ee19c0366d829e291451be638e59,239634"

6 "0x390de26d772d2e2005c6d1d24afc902bae37a4bb,229565"

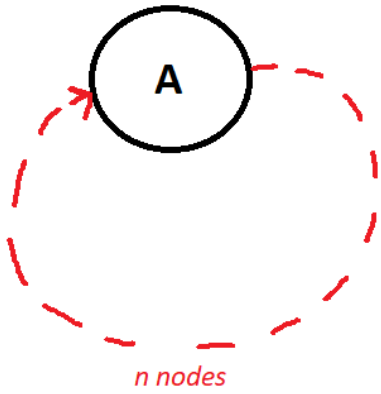
7 "0x2b5634c42055806a59e9107ed44d43c426e58258,206157"

8 "0x6cc5f688a315f3dc28a7781717a9a798a59fda7b,176553"

9 "0x75e7f640bf6968b6f32c47a3cd82c3c2c9dcae68,165784"

10 "0x69ea6b31ef305d6b99bb2d4c9d99456fa108b02a,123404"

0x75e7f640bf6968b6f32c47a3cd82c3c2c9dcae68 (appearing at rank 9) is the most frequently wash traded address, with the cycles totalling to 852238. Some of these cycles may be like item d in figure 4, so the number of cycles may be a little bit lower however, it is clear that 0x75e7f640bf6968b6f32c47a3cd82c3c2c9dcae68 has the most wash trades. The addresses from 1-8 belong to well known crypto exchanges according to Etherscan and so cannot be associated with wash trades.



However, this can be a little bit inaccurate as there is no clear picture of the chains or the cycle itself, as the graph the criteria is looking for now looks something akin to the following graph:

the job does not know what other nodes exist, so a potential improvement to the algorithm is to evaluate the complexity of the loops.

Contract Types

To identify hidden contract types, we would have to utilise a machine learning technique called clustering to try to discover groupings of contracts that can be discovered. To do this we would first need to extract the features to group.

I selected the following as the features to get a potential clustering of values:

average gas price, average gas, block number, total value

to run the file we would do:

```
python Contracts2.py -r hadoop
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/transactions
hdfs://andromeda.eecs.qmul.ac.uk/data/ethereum/contracts > features
```

I chose these because they would be representative of a contract as a whole and allow separation based on how far these features differ but how closely they can relate. Gas price and gas could have some of correlation and in conjunction with total value and block number it would further separate said groups.

I retrieved these features by running the map reduce code in ContractFeatureExtractor.py, this functions by joining transactions with contracts in the mapper and in the reducer calculates the average values and the total values. More information is provided in the ContractFeatureExtractor.py file. The output was in an unsuitable format so there is a converter

script to make a comma separated variable file out of the mapreduce output, this ode is in the CSVConvert.py file and it finally produced features.csv out of features.txt. A sample of these features are as follows:

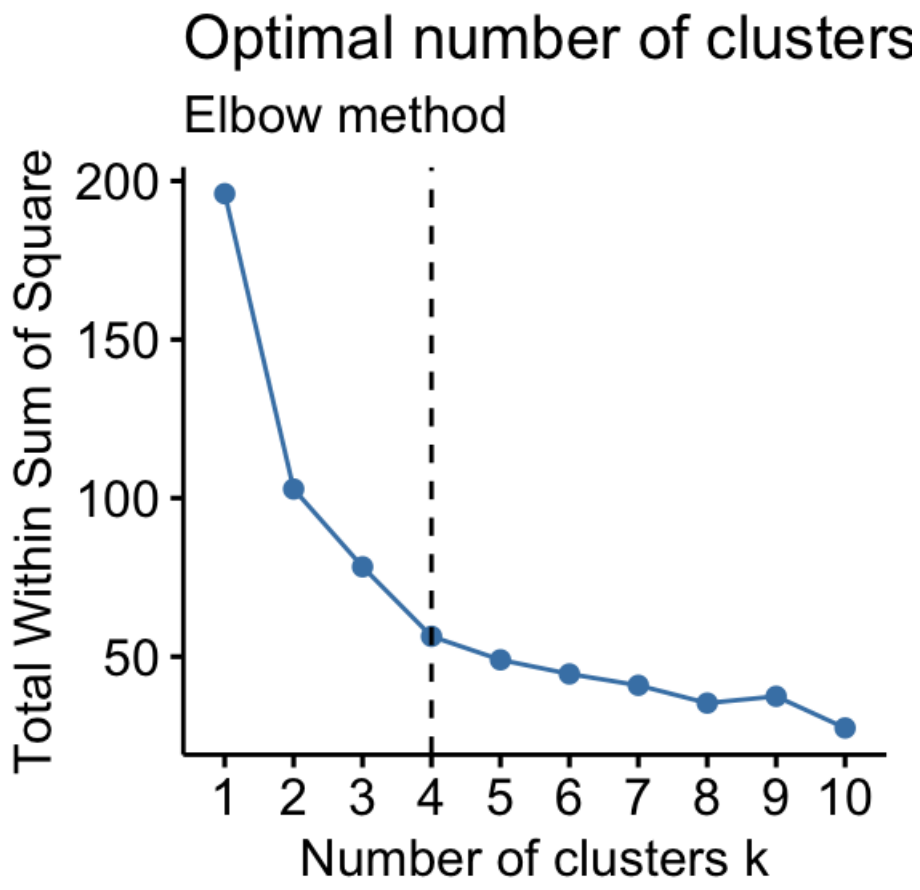
0x0000c48b539e783537923da2b635ef202a9ffc2a	1E+10	50000	7410312	2.61E+17
0x0000de3c4a8daf93dbe5c4b28aeb486236b82da8	6.39E+09	52834.65	4232987	1.47E+19
0x00012df38ea3a6dabefb8407a59219a0c7dd0bc8	4.1E+10	21280	6542899	1E+18
0x0002d1bd02db58653555c8cca521dab59279b923	5E+10	21051	5020358	2.32E+18
0x000313379d160d984156c54bf48597e21b0dc85b	8E+09	25246	5041350	3E+16

From these features we can move on to the kmeans algorithm. This is done using spark's MLib library.

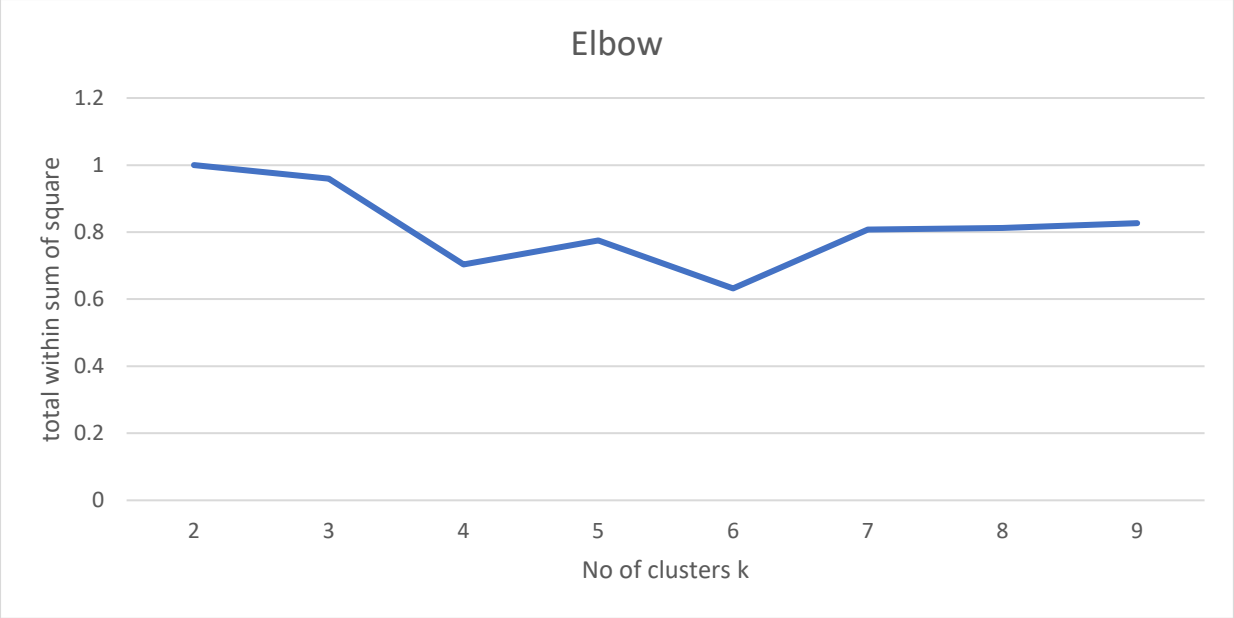
To run the kmeans algorithm we would run:

```
spark-submit kmeans.py
```

First we would normalise the data, here we used a standard scaler which we used to calculate the silhouette score to output an elbow graph which would show the optimal number of different classifications for contract types. The graph should be a downward smooth slope and the optimal number of classifications would be the "elbow".



However the graph we received from our results looked like:



We can try to extrapolate that the number of clusters is 6 as this is the lowest point of the elbow, but this is likely not the case and the normalisation algorithm was likely not good enough.

Miscellaneous Analysis

1. Fork the Chain

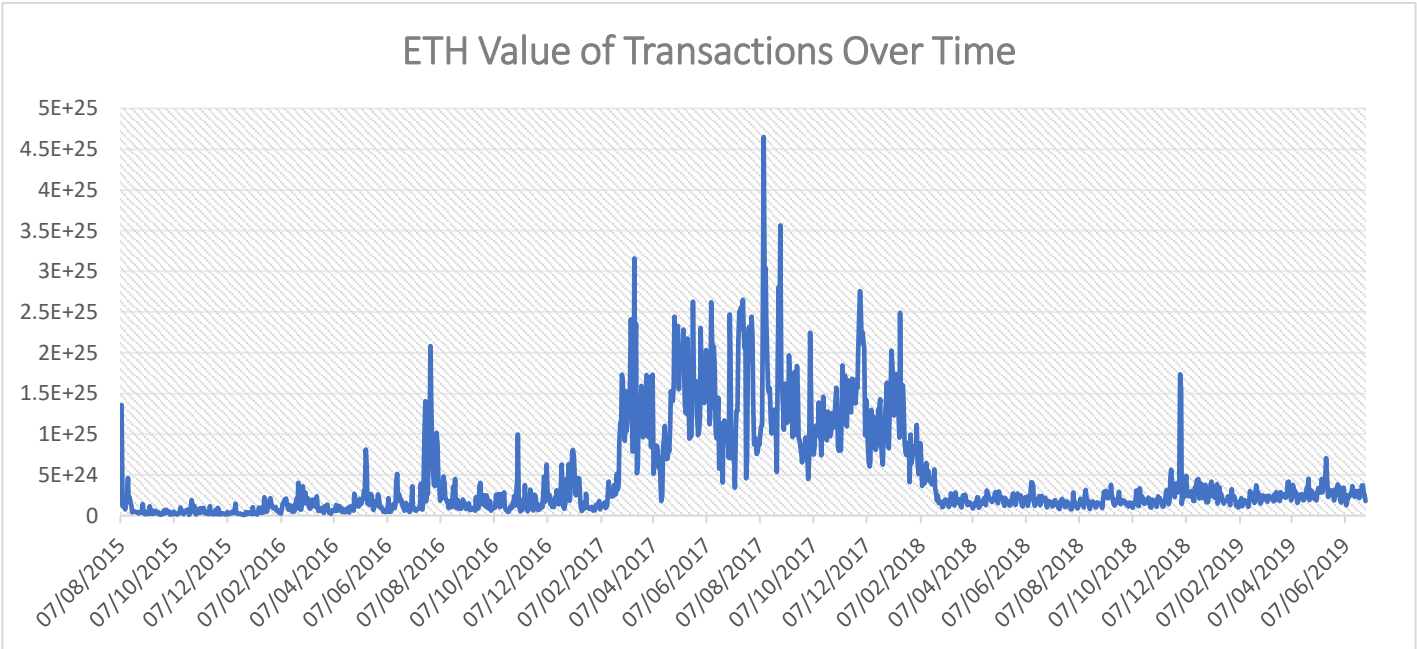


Figure 5 - the value of ETH transactions daily

There were some dates that followed directly after certain Ethereum forks (such as Constantinople) that did not appear in the dataset produced. However, there are dates we have directly following after the Byzantium fork which occurred at Oct-16-2017 05:22:11 AM +UTC (Ethereum Team, 2021)

Run by:

forkTheChainTopGainersByzantium.py –

```
spark-submit forkTheChainTopGainersByzantium.py > top10GainersByzantium.csv
```

forkTheChainTransactionsPerDay.py –

```
spark-submit forkTheChainTransactionsPerDay.py > totalByTime.csv
```

16/10/2017	7.41E+24
17/10/2017	1.11E+25
18/10/2017	1.46E+25
19/10/2017	1.12E+25
20/10/2017	1.14E+25
21/10/2017	1.04E+25
22/10/2017	9.26E+24
23/10/2017	1.28E+25
24/10/2017	1.25E+25
25/10/2017	1.02E+25

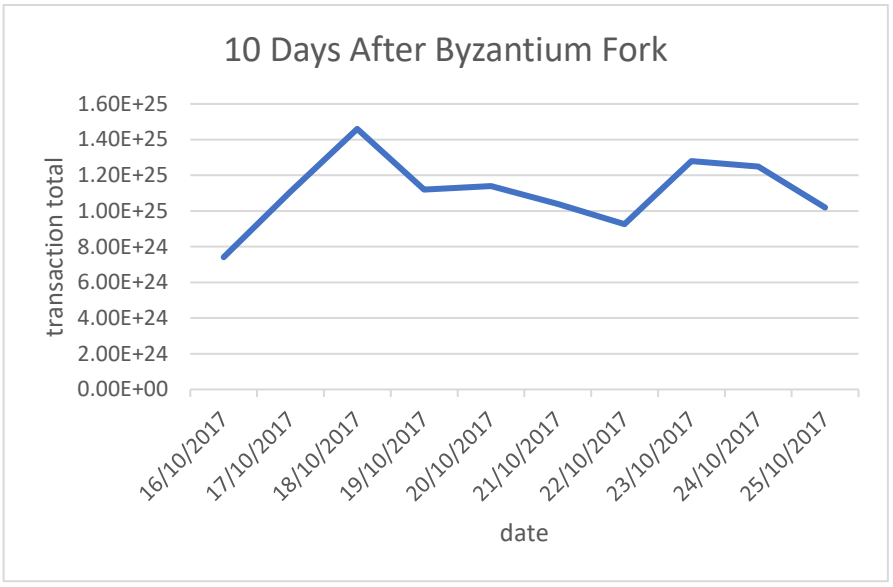


Figure 6 - transaction totals per day after Byzantium fork

The values show a clear spike from a value of 7.41E+24 Wei to 1.11E+25 Wei following the release of the new fork. A magnitude bigger than the E+24 in the previous day showing that speculation was for Ethereum's growth. This is likely to do with what this fork brought with it.

> "
> The Byzantium fork:
>
> Reduced block mining rewards from 5 to 3 ETH.
> Delayed the difficulty bomb by a year.
> Added ability to make non-state-changing calls to other contracts.
> Added certain cryptography methods to allow for layer 2 scaling.
> " (Ethereum Team, 2021)

These quality-of-life improvements most likely lead to the future of ETH looking promising for blockchain developers, especially with the ability to call other contracts in a non-state-changing manor and layer 2 scaling as well.

The main gainers of this increase of Ethereum price are the addresses:

0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be	8.21E+24
0x7727e5113d1d161373623e5f49fd568b4f543a9e	6.76E+24
0x876eabf441b2ee5b5b0554fd502a8e0600950cfa	5.85E+24

0x22b84d5ffea8b801c0422afe752377a64aa738c2	4.25E+24
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3	3.92E+24
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8	3.78E+24
0xc257274276a4e539741ca11b590b9447b26a8051	3.69E+24
0xf4b51b14b9ee30dc37ec970b50a486f37686e2a8	3.34E+24
0xfa52274dd61e1643d2205169732f29114bc240b3	2.67E+24
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef	1.98E+24

The amount gained beside them was calculated by aggregating the total amount of transactions to that address in that month and the following 3 months (the Byzantium fork occurred in October, so the cut-off month was January 2018), this was to capture potential runoff revenue from the fork that may not be captured otherwise. However, this comes with a degree of unreliability because there may be other factors affecting the revenue to these addresses.
The addresses belong to the following holders: 1. Binance 2. Bitfinex 3. Bitfinex 4. ETH Classic 5. Bittrex etc.

Another fork that we have some information on is the Tangerine whistle fork, which the value of transactions changed like so:

18/10/2016	2.8634E+24
19/10/2016	1.06111E+24
20/10/2016	7.26631E+23
21/10/2016	6.2015E+23
22/10/2016	5.0185E+23
23/10/2016	4.32875E+23

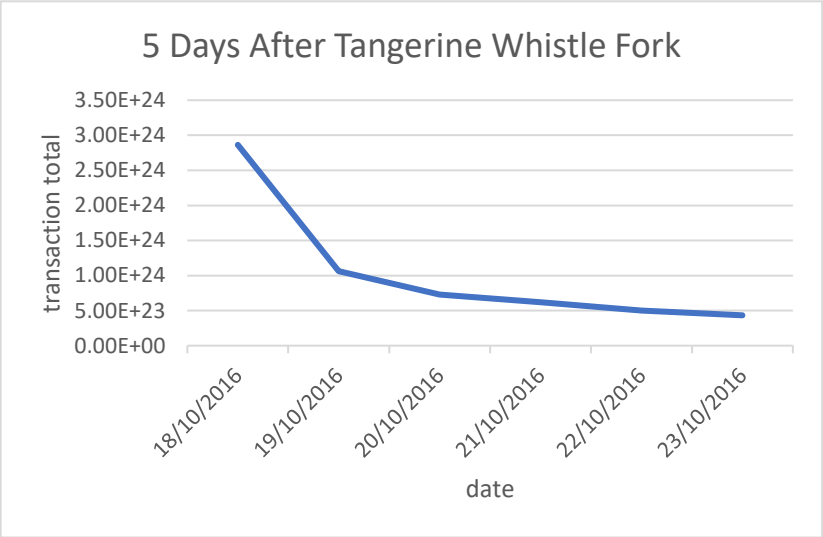


Figure 7 - transaction totals per day after Tangerine Whistle fork

This dip in the value of transactions can be assumed to also be a dip in the amount of transactions on the Ethereum blockchain at that time. We can gain more insight into why this is by looking at the tangerine whistle summary.

> “
> The Tangerine Whistle fork was the first response to the denial of service (DoS) attacks on the network (September/October 2016) including:
>
> addressing urgent network health issues concerning underpriced operation codes.
> ” (Ethereum Team, 2021)

In response to a DoS attack on the network, the Ethereum team pushed out this fork to try to mitigate this. This left a lot of people unsure about the network and the future prospects of Ethereum which more than likely reduced the number of transactions by that amount (reducing the value of transactions by over half from the 18th to the 19th).

Following the Tangerine Whistle fork, we have some information on is the Spurious Dragon fork in which the value of transactions changed as follows:

22/11/2016	2.58915E+24
23/11/2016	2.43788E+24
24/11/2016	8.39942E+23
25/11/2016	1.52045E+24
26/11/2016	7.03598E+23
27/11/2016	7.12635E+23
28/11/2016	7.97084E+23
29/11/2016	1.31809E+24

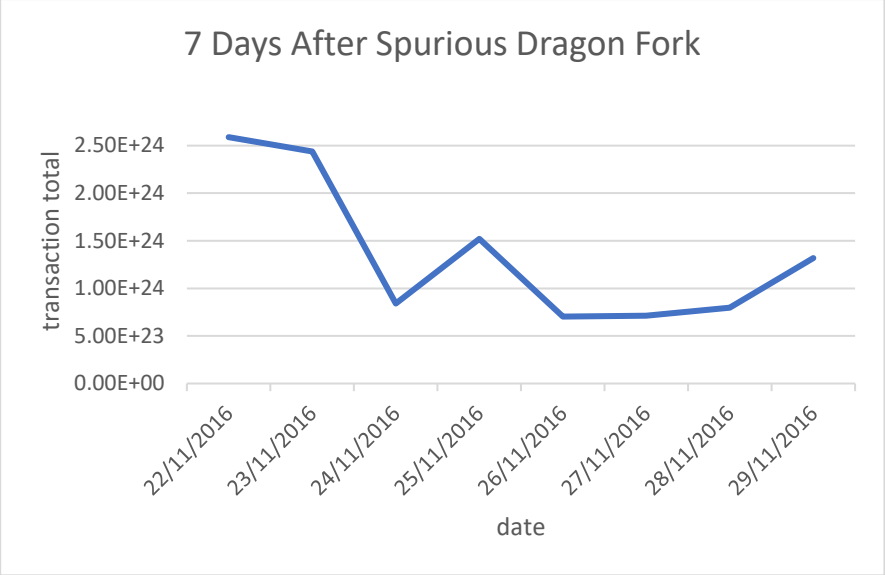


Figure 8 - transaction totals per day after Spurious Dragon fork

After this fork, the number of transactions once again declined. This is because the Spurious Dragon fork was a secondary response to the DoS attack and like the Tangerine Whistle fork before it: this fork reduced the value of transactions again**.**

2. Comparative Analysis

With this job I recreated the code done in part B (to find the top 10 most popular contract services) but using spark. This was done in 2 ways:

1. By transforming the RDD into a dataframe to compute the join
2. By joining the RDD with another RDD

To run the timers:

Btimer.py –

```
python Btimer.py
```

partB-spark-timer.py –

```
python partB-spark-timer.py
```

For simplicity, I'll be using my initial attempt of using dataframes for the comparison between the mapReduce code. The output to both is slightly different but this was because of the large numbers and how they are represented and evaluated. The outputs to both are as follows:

rank **address**, **aggregate amount**

Map-Reduce Output	Spark Output
1"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,841551008009965865822726776"	1,0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,8.415510081e+25
2"0xfa52274dd61e1643d2205169732f29114bc240b3,45787484844831893529864788805"	2,0xfa52274dd61e1643d2205169732f29114bc240b3,4.57874844832e+25
3"0x7727e5113d1d161373623e5f49fd568b4f543a9e,4562062400135071255726857 3"	3,0x7727e5113d1d161373623e5f49fd568b4f543a9e,4.562062400 14e+25
4"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4317035609226246891929896 9"	4,0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4.31703560923e+25
5"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,270689215820195424998828 77"	5,0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,2.7068921582e+25
6"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,211041951380936600500000 00"	6,0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,2.11041951381e+25
7"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,155623989568026802112254719409"	7,0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,1.55623989568e+25
8"0xbb9bc244d798123fde783fcc1c72d3bb8c189413,119836087292028938468186 81"	8,0xbb9bc244d798123fde783fcc1c72d3bb8c189413,1.19836087292e+25
9"0xabbb6bebfa05aa13e908eaa492bd7a8343760477,117064571779408955217704 04"	9,0xabbb6bebfa05aa13e908eaa492bd7a8343760477,1.17064571779e+25
10"0x341e790174e3a4d35b65fdc067b6b5634a61caea,837900075191775562405750 0"	10,0x341e790174e3a4d35b65fdc067b6b5634a61caea,8.37900075192e+24

The results are identical, now the time taken to achieve this result is a different matter. To achieve this timing I used a python script that would time from the start of execution to the end. This would include the waiting time for the cluster to accept the job and run it (both on spark and mapreduce), which in terms of comparison is fair and a balanced review. Although these timings are subject to an unknown variable: the number of people currently using the cluster at any one time and so the environment is not completely the same an it will be difficult to achieve a totally fair test.

The results for the timing script for spark is as follows:

1,0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,8.415510081e+25
2,0xfa52274dd61e1643d2205169732f29114bc240b3,4.57874844832e+25
3,0x7727e5113d1d161373623e5f49fd568b4f543a9e,4.56206240014e+25
4,0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4.31703560923e+25
5,0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,2.7068921582e+25
6,0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,2.11041951381e+25
7,0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,1.55623989568e+25
8,0xbb9bc244d798123fde783fcc1c72d3bb8c189413,1.19836087292e+25
9,0xabbb6bebfa05aa13e908eaa492bd7a8343760477,1.17064571779e+25
10,0x341e790174e3a4d35b65fdc067b6b5634a61caea,8.37900075192e+24

---

attempt 1

---

spark run: 378.347514153

1,0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,8.415510081e+25
2,0xfa52274dd61e1643d2205169732f29114bc240b3,4.57874844832e+25
3,0x7727e5113d1d161373623e5f49fd568b4f543a9e,4.56206240014e+25
4,0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4.31703560923e+25
5,0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,2.7068921582e+25
6,0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,2.11041951381e+25
7,0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,1.55623989568e+25
8,0xbb9bc244d798123fde783fcc1c72d3bb8c189413,1.19836087292e+25
9,0xabbb6bebfa05aa13e908eaa492bd7a8343760477,1.17064571779e+25
10,0x341e790174e3a4d35b65fdc067b6b5634a61caea,8.37900075192e+24

---

attempt 2

---

spark run: 36.8383588791

1,0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,8.415510081e+25
2,0xfa52274dd61e1643d2205169732f29114bc240b3,4.57874844832e+25
3,0x7727e5113d1d161373623e5f49fd568b4f543a9e,4.56206240014e+25
4,0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4.31703560923e+25
5,0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,2.7068921582e+25
6,0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,2.11041951381e+25
7,0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,1.55623989568e+25
8,0xbb9bc244d798123fde783fcc1c72d3bb8c189413,1.19836087292e+25
9,0xabbb6bebfa05aa13e908eaa492bd7a8343760477,1.17064571779e+25
10,0x341e790174e3a4d35b65fdc067b6b5634a61caea,8.37900075192e+24

---

attempt 3

---

spark run: 27.1659917831

1,0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,8.415510081e+25
2,0xfa52274dd61e1643d2205169732f29114bc240b3,4.57874844832e+25
3,0x7727e5113d1d161373623e5f49fd568b4f543a9e,4.56206240014e+25
4,0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4.31703560923e+25
5,0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,2.7068921582e+25
6,0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,2.11041951381e+25
7,0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,1.55623989568e+25
8,0xbb9bc244d798123fde783fcc1c72d3bb8c189413,1.19836087292e+25
9,0xabbb6bebfa05aa13e908eaa492bd7a8343760477,1.17064571779e+25
10,0x341e790174e3a4d35b65fdc067b6b5634a61caea,8.37900075192e+24

---

attempt 4

---

spark run: 25.8763990402

---

average time: 97.9991493702

This says that the spark jobs had an average run time of 97.9991493702 seconds, but each run got significantly faster. This may be because of the scheduler taking the most time to set up the job on first run and having less trouble with the following jobs. The speed increase could not be from RDD caching because there was no use of persist or cache in any of the jobs.

---

attempt 1

---

map reduce run: 2467.6540446281433

---

attempt 2

---

map reduce run:: 2467.053475379944

---

attempt 3

---

map reduce run:: 2705.021505355835

---

attempt 4

---

map reduce run:: 3050.3008110523224

---

average time: 2138.005967283249

However with mapreduce, I replicated the same timing solution however the time difference was very big. With the average time for a map reduce job taking 2138 seconds or 35.63 minutes. This is an almost 35x time difference between spark and mapreduce jobs.

This means that spark is the clear victor and the correct choice for the job.

3. Gas Guzzlers

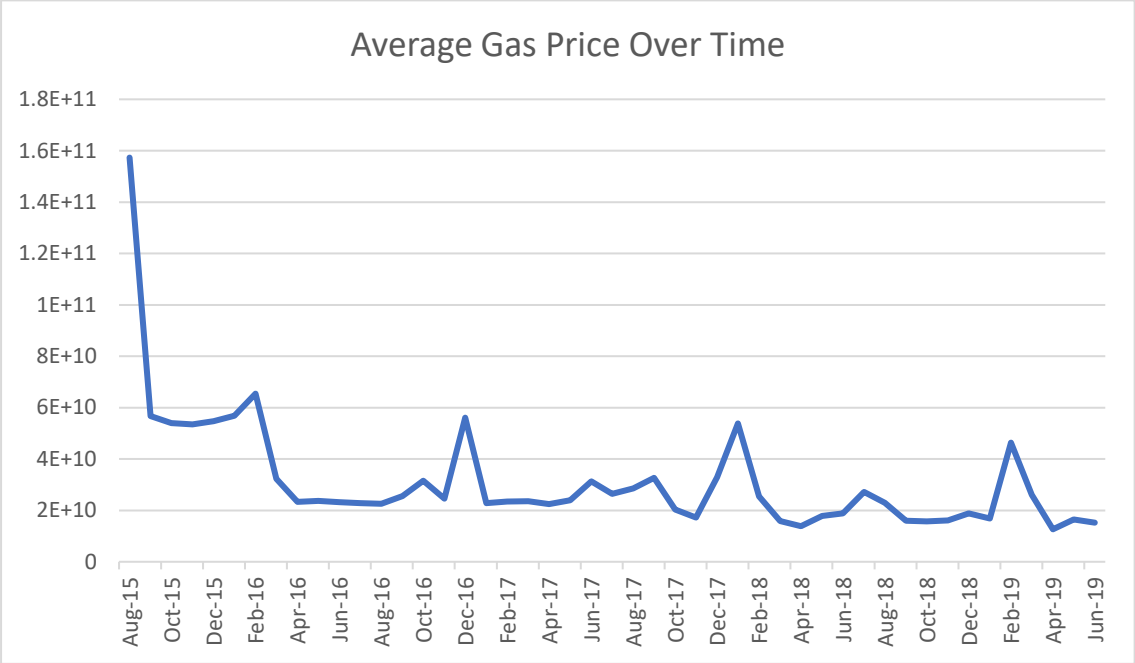
To Calculate the way gas has changed over time, there would first need to be a calculation of the average gas price overtime (per month) to build up a broader idea of gas prices.

To run this:

gasGuzzlers.py –

```
spark-submit gasGuzzlers.py > gasChangeMonth.csv
```

The following is the graph per month of gas prices:



This change in average gas price shows that from Ethereum's inception: the price of gas per transaction has gone down until it reached an equilibrium with a few jumps in value in December 2016, December/January 2017 and February 2019.

References

cryptoinfowatch. (2021, october 17). *333 ETH Review: Is This Project A Scam?* Retrieved from cryptoinfowatch: https://www.cryptoinfowatch.com/333-eth-review-is-this-project-a-scam/

Ethereum Team. (2021). *Ethereum History of Forks*. Retrieved from Ethereum.org: https://ethereum.org/en/history/

Meears, A. (2018, October 2). *333ETH Flagged as Scam by MetaMask*. Retrieved from fullycrypto: https://fullycrypto.com/333eth-flagged-as-scam-by-metamask

OmiseGO. (2021). *OMG Network*. Retrieved from OMG Network: https://omg.network/