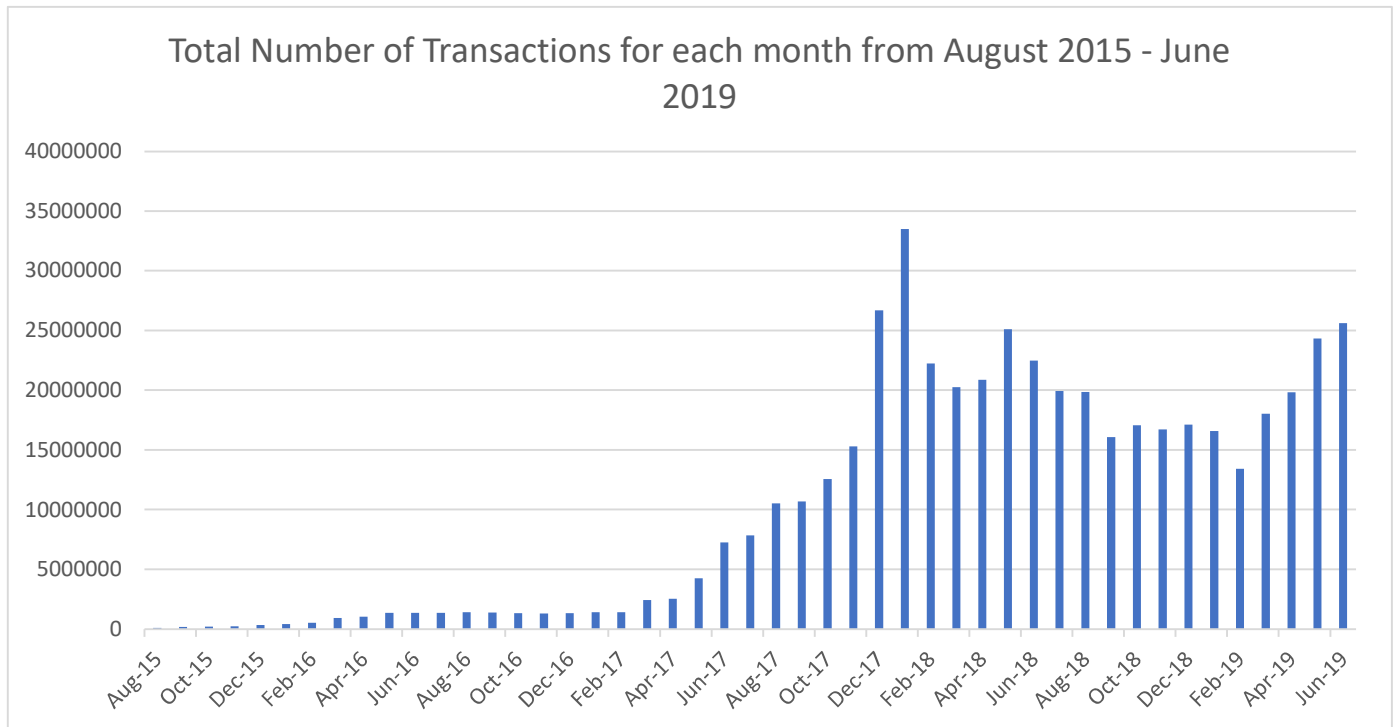


Ethereum Analysis Coursework

Part A: Time Analysis

Total Number of Transactions

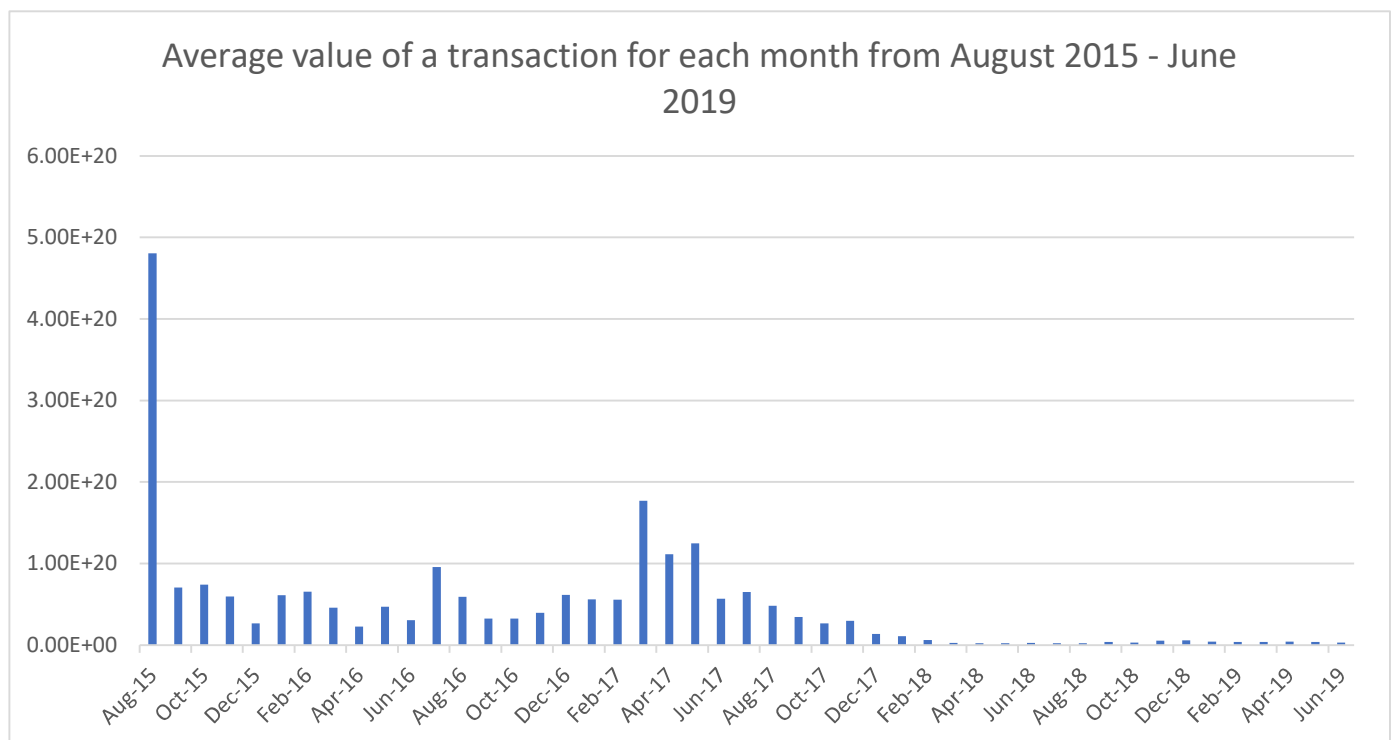


In order to work out the total number of transactions, in my MapReduce program I had a mapper which would simply take the Unix timestamp from the fields and convert it into a Month and Year format. This would be the key and the value that would be passed would be simply a one. This is because I am working out the total number of transactions for each month for each year and so I will simply add it up.

I added a combiner in my program in order to reduce the amount of data that would enter the reducer. The combiner is like a reducer, but it takes place after the mapper. The combiner would perform the same task as the reducer, which was to get the sum of the values outputted by the mapper and group it with the associated time. The combiner once finished will then output the results to the reducer.

The reducer will perform the very same task as the combiner. The reducer will sum up values into a single value and group these values with the associated key just like the combiner but will group all the values to the keys associated with it. The reducer will then output these results for the user to see giving the number of transactions for the given time period with the output plotted on the graph above.

Average Value of Transactions



For the second task, this time I had to get the average value of a transaction for each month given a year. This was not the same as the first task, as this time I had to work out the average. The numerical summarisation path was taken in order to achieve this. In my MapReduce program, what I did was the almost the same as the first task, only this time I also got the value for each transaction. My mapper would return the time as the key once again and this time as the value a tuple containing the transaction value and the number 1 would be returned. This is because for the combiner and reducer these values are important for me.

Combiner and Reducer perform the same task, with the exception being that the combiner would return the total number of transactions for a given time and the sum of the values of those transactions in a tuple, and the reducer would calculate the actual average with those values. Everything else remains the same. The combiner and reducer create two variables, count and total. Count is used to calculate the sum of the values and total is used to work out the total number of transactions. The timestamp will be the key and the values will be the list of tuples containing the value and 1. A for loop is made going through the variable values which contains the list of tuples. What happens in each increment in the loop is that count will take the first element of the selected tuple and value will take the second element of the selected tuple and add them into their respective variables. At the end of the combiner, the combiner will return the timestamp and a tuple containing the sum of the values of the transactions and the total number of transactions.

In the reducer, the same task as the combiner is performed, but the difference is that whilst the time will still be returned as the key, the value outputted is different, where the average is worked out with the values of count and total by doing $\text{count} / \text{total}$ in the reducer, unlike the combiner where a tuple was sent containing the two variables.

The results were then outputted and then plotted on the graph above.

Part B: Top 10 Most Popular Services

"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	84155100809965865822726776
"0xfa52274dd61e1643d2205169732f29114bc240b3"	45787484483189352986478805
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"	45620624001350712557268573
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"	43170356092262468919298969
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"	27068921582019542499882877
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"	21104195138093660050000000
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"	15562398956802112254719409
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"	11983608729202893846818681
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"	11706457177940895521770404
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"	8379000751917755624057500

This operation comprised of two jobs. The first job involved both the aggregation and joining processes. In the mapper of the first job, an if statement would determine whether the line came from the contracts dataset or transactions dataset. 5 fields after the split would be from the contracts dataset and 7 fields after the split would be from the transactions dataset. If the line was from contracts, the address is only needed. The address would be sent as the key and the value would be a tuple that would have 2 elements which were both the integer 1, one of which is just a filler to match the number of elements in the tuple. If the line came from the transactions dataset, we are looking for two things. The address and the value of the transaction. The address will be the key and the value of the transaction will be one of the values in the tuple that is to be sent as the value. The other value in the tuple would be the integer 2. This will be yielded to the reducer task. The integers 1 and 2 as the second elements are needed for the join.

When the key value pairs are sent to the reducer, the reducer will aim to send to the next job the transactions in which the addresses are found in both datasets. The addresses that are found in both datasets are sent whereas the addresses which are not in both are discarded. In order to perform this, an array will hold the values for a particular address, and a variable containing a boolean will state whether the address exists in both datasets. This is where the second element of the tuple come into play. If the second value is 1, then it is the address that came from the contracts dataset and because it exists, this will be set as True. If the second element is 2 however, this is from the transactions dataset, and what happens is that the value of the transactions that was sent by the mapper gets added into the array. Upon the conclusion of the task, an if statement will see whether the address existed in both datasets. We use the array to sum the values together in order to obtain the aggregated value for a given time. If the sum of the values in the array was greater than 0 AND the variable holding our boolean is true, then we will send this address and the sum of the values. The address will be the key and the sum will be the value, this is sent to the mapper of the next job since we have to perform the top 10 operation. If an address does not meet the two conditions, the address will not be passed on.

The mapping task of the top 10 job is relatively straightforward, as it simply just passes the values to the combiner. However, there is no key that is sent, as we are passing both the address and the sum of the values in a tuple as the value.

The combiner and reducer for the second job will perform the same task which is to get the top 10 highest values. The values that get passed down get sorted based on the size of the values and is sorted from highest to lowest. A for loop after the sorting will then return only the top 10 elements of the sorted array, the address and sum is passed once again as a value in the combiner with the

key being 'top' as just a filler for the key. The reducer will receive these values and will pretty much perform the same task, but this time when outputting it will output the address as the key and the sum as the value. The for loop will output the top 10 contracts with the highest value.

Part C: Top 10 Most Active Miners

"0xea674fdde714fd979de3edf0f56aa9716b898ec8"	23989401188
"0x829bd824b016326a401d083b33d092293333a830"	15010222714
"0x5a0b54d5dc17e0aad383d2db43b0a0d3e029c4c"	13978859941
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5"	10998145387
"0xb2930b35844a230f00e51431acae96fe543a0347"	7842595276
"0x2a65aca4d5fc5b5c859090a6c34d164135398226"	3628875680
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01"	1221833144
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb"	1152472379
"0x1e9939daaad6924ad004c2560e90804164900341"	1080301927
"0x61c808d82a3ac53231750dad13c777b59310bd9"	692942577

The operation was performed with 2 jobs. The first job dealt with gathering the total size of the blocks mined with each miner. This job was pretty much the same as the word count problem. In the mapper, the miner and size of the block was taken as the values and the mapper would return the miner as the key and the size as the value. However, this would only be returned if the size of the block was greater than 0.

Combiner and reducer both did the same task. They would simply work out the sum of the size of the blocks mined for each miner and return it. The combiner would send this data to the reducer, the miner being the key and the sum of the sizes as the value. The reducer would receive this data and would do the same thing. The reducer will send the data to the mapper of the top 10 job.

The mapper of the top 10 job doesn't do much here, it just simply sends the data it receives from the reducer of the previous job and sends it to the combiner of the current job. However, it doesn't send a key. It will send only a value which comprises of a tuple that contains both the miner and the sum of the sizes for that miner.

The combiner and reducer will take the tuples and then sort it in terms of the largest sizes. At this point we can easily get our top 10. The tuples get sorted based on the second element which is the sum of the sizes. As reverse is equal to True, instead of doing it from smallest to largest, it will sort it from largest to smallest. The for loop will then begin where the top 10 miners will be outputted. The for loop will simply return the first 10 values from the new sorted list of tuples and will stop once rank reaches 10.

The combiner when returning will return "top" as the key as just a filler. The value will be the tuple containing the miner and the sum of the sizes. The reducer however instead of placing them both in the values section, will place the miner as the key and the sum of the sizes as the value. The result is then outputted.

Part D: Miscellaneous Analysis

Scam Analysis

In order to find the most lucrative form of scam, the category that gained the most profit had to be obtained. 2 tasks were performed involving an aggregation of the values and ranking the most profitable categories from highest to lowest.

The scams.json file was converted into a csv file in order to make things easier to code.

A MapReduce task was performed in which there were three jobs. The first job was to join the two tables together. The type of join that was done was a replication join, as the scams.csv was small enough to fit in memory. There is no reducer for this task. The mapper_init task would grab the necessary fields needed for the join, which is in this case is the scam address and the category.

The mapper task would perform the replication join, where the addresses in the transactions dataset and the scams.csv file would be joined. If the addresses match, then the respective value and category of the scam will be yielded to the next job, which will perform the aggregating task. The key will be the category and the value will be the value of the associated transaction.

The mapper of the aggregation job will simply just pass on the key and value pair to the combiner, it does nothing else.

At the combiner, what it will do is that when sending the data to the reducer, it will send the category and the sum of the values that are associated with the category.

The reducer will perform the very same task and will send the final results to the mapper of the ranking task.

The ranking job will receive these outputs and will then output them to the combiner. The category and value will be sent as the value as a tuple, no key will be outputted to the combiner of the job.

The combiner will sort the tuples from biggest to smallest in terms of the value. The tuples are then sorted based on the second element which is the sum of the values. Reverse is True, so it will be sorted from largest to smallest. The for loop will then begin which will send the data. The key sent is "top" which is just a filler, and the value will be the tuple of the category and sum of the values.

The reducer will do the same task as the combiner. At the end of the reducing task when the data is outputted, the key will be the category and the value will be the sum of the values for that category. The final results will be outputted to the machine as this is the final job of the MapReduce task.

The following results were outputted:

```
"Scamming"      38407781260421703730344
"Phishing"      26927757396110618476458
"Fake ICO"      1356457566889629979678
```

We can see here that the most lucrative form of scam is the Scamming category. We will use this category for further analysis on how much transactions it had through its period.

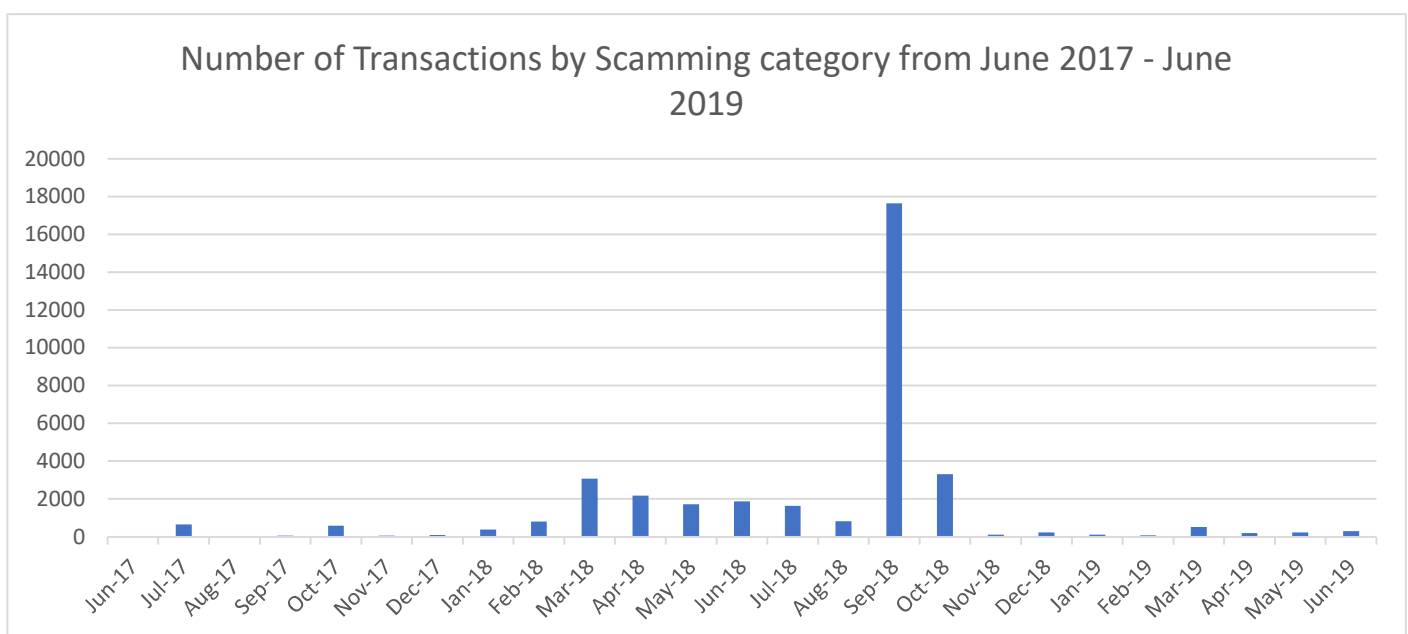
The next step is to now work out how this category has changed over time, so to do this the number of times a transaction throughout the whole time period will be done.

To get this output, virtually the same task from getting the most lucrative form was performed with the exception of adding the ranking task, meaning that there were only two tasks.

The first job was pretty much the same, a replication join was performed to get the associated values and categories of scam for the address, only this time we only want the ones that were associated with the "Scamming" category. An if statement will determine whether the transaction belongs to the "Scamming" category. What would be sent is the timestamp of the transaction which was converted into a month and year based on the timestamp. This was sent as the key and the integer 1 was sent as the value. The value 1 will be used to aggregate the total number of transactions that were performed to the addresses associated with this form of scam for each given time.

The second join remained identical, where it simply aggregated the values and grouped them by month and year. The mapper of the aggregation job will just pass on the key and value pair to the combiner. The combiner will send the given month and year and the sum of the values that are associated with it to the reducer, and the reducer will do the very same thing but this time outputs the results.

The output that was obtained was then plotted in a bar graph to get the following result:



Based on the result given, this category has very little transactions up until around 2018. At this point, the number of transactions begin to increase gradually. Between March 2018 to August 2018 the number fluctuates until September 2018 where the number of transactions to addresses associated with this category skyrockets to over 16000 transactions in that month. Looking at the graph, September 2018 is the peak of this category. The amount then decreases further on at this point, with October 2018 having a similar amount as March 2018, and the further months having dropped drastically.

This drastic drop may have a correlation with various scams going offline or inactive. To confirm this, the `scams.csv` file was looked at the activities for the Scamming category were looked to see whether they were offline, active, etc. The results obtained will let us know if these categories have been going offline as time passed on.

As the `scams.csv` file was used which was on my local files, the Hadoop cluster was not used. Instead, MapReduce was run locally.

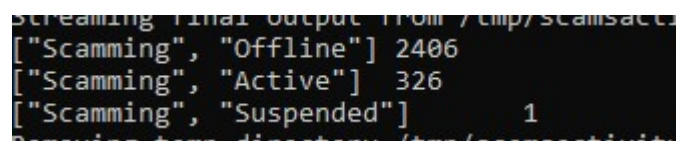
This was a two job process, with an aggregate step running first then the ranking job running after. The mapper of the first job will simply take the category which in this case is Scamming and the activity of the scam. This will enter an if statement whether the category is Scamming or not. If it is, then it is sent to the combiner, if it isn't then it isn't sent. This is because we are looking at the most lucrative scam category and checking the activity for its scams to see if that has something to do with the drop of the previous graph. The key sent will be a tuple containing the category and the activity, with the value being 1 since we are getting the total number of scams for each activity.

The combiner will just sum up the values into one single value and group these values to its associated key. This will get sent to the reducer, which does the exact same job as the combiner. The reducer will output this and send it to the mapper of the ranking job.

The mapper of the next job will take the input and will just send it to the combiner. There will be no key sent, and the value section will contain both the tuple containing the category and activity as well as the aggregated value.

The combiner will receive this and will sort the values it gets from highest to lowest based on the aggregated values. The tuples are sorted based on the aggregated values and a for loop will simply output the tuple and aggregated value as the value of the pair. The reducer will perform the exact same task but when outputting, the tuple will sent as the key and the aggregated value will be the value for the pair. The results are then outputted.

The following output obtained was found:



```
streaming final output from /tmp/scamsact1
["Scamming", "Offline"] 2406
["Scamming", "Active"] 326
["Scamming", "Suspended"] 1
removing temp directory /tmp/scamsactivity
```

This output shows that there is a correlation as to why the activity of the scams have dropped. As depicted on the screenshots, the offline category is significantly larger than the active category meaning that they are no longer able to receive ether, resulting in a drop of profit. This data was based on June 2019 which is the last month for the whole dataset in general.

Fork the Chain

The Byzantium fork in October 2017 was looked at as it would give me a good representation of the general usage and price of Ethereum from before the fork and after the fork. I used August 2015 to October 2017 up until the fork happened for analysing information before the fork and the remaining dates for after the fork.

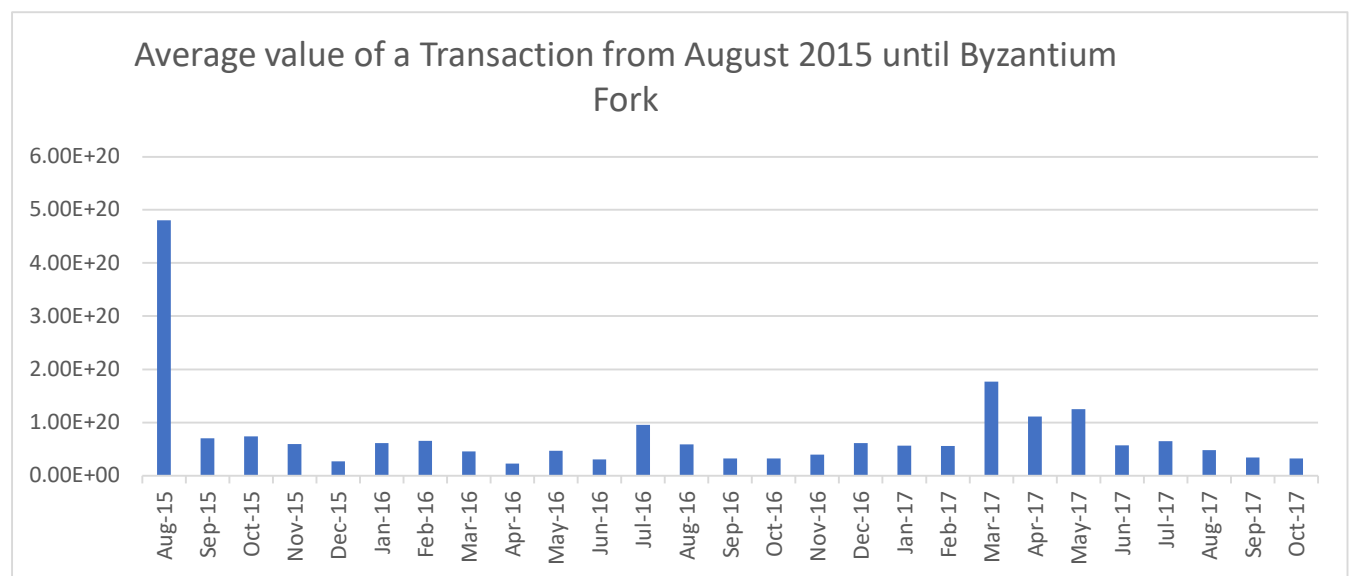
The first few jobs were to get the average price of a transaction over time before the fork and after the fork. This is pretty much the very same task as the second job of Part A, which is to get the average price of a transaction over time. Only this time you would only go up until the timestamp of the fork.

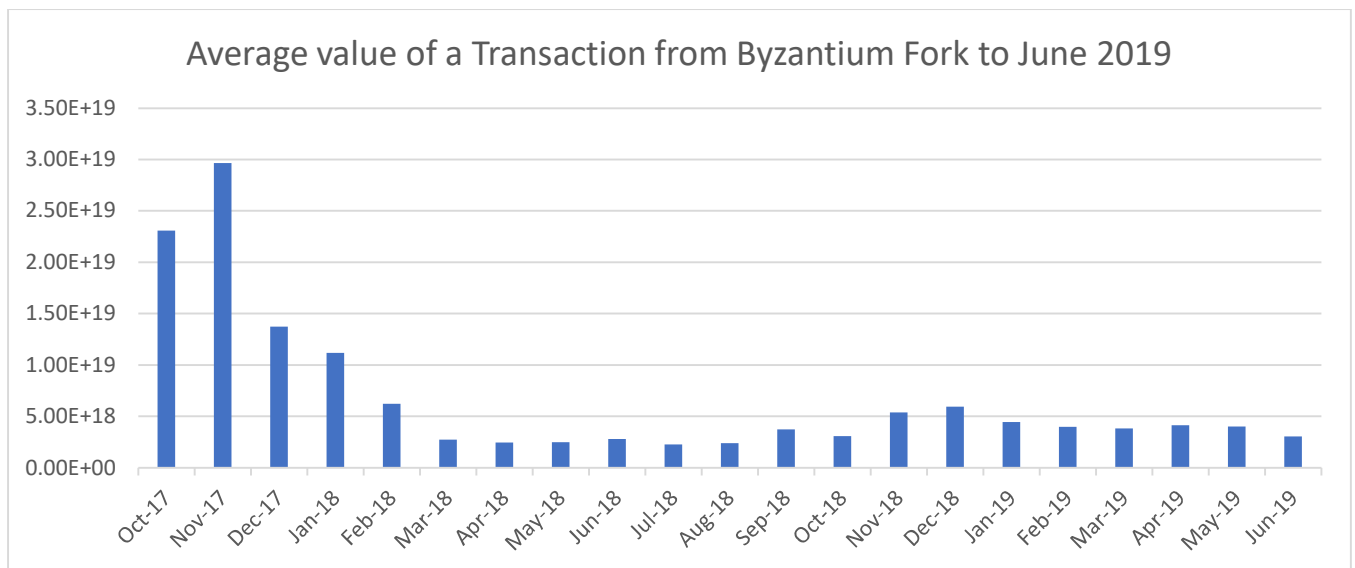
My mappers would return the time as the key and tuple containing the transaction value and the number 1 was returned as the value. These values will be needed for the combiner and reducer. A timestamp will be taken which will determine whether the transaction was before the fork or not. If it was the job focusing on before the fork, if the timestamp was smaller than the fork timestamp then this is before the fork and will be used. If the transaction timestamp is greater than the fork timestamp, then the transaction will not be used. For the other job focusing on after the fork, this would be the other way round.

Combiner and Reducer perform the same task, difference being that the combiner would return the total number of transactions for a given time and the sum of the values of those transactions for that time, and the reducer would calculate the actual average for it. The combiner and reducer create two variables, count and total. Count is used to calculate the sum of the values and total is used to work out the total number of transactions. A for loop will take the first element and second element of the selected tuple and add them into their respective variables. The combiner will return the timestamp and a tuple containing the sum of the values of the transactions and the total number of transactions whereas for the reducer the timestamp will still be returned as the key, but the average value will be worked out at the end.

These jobs had the same code, the only difference being the focus on whether getting the transactions before or after the fork.

Following the finishing of the two related jobs, the following graphs were plotted based on their respective outputs:





As we have got the two graphs for the average value of a transaction before and after the fork, we can now compare the two graphs to see if there are any changes. We can see that the average value of a transaction after the fork has actually decreased, meaning that there has been a price drop of Ethereum. The price was much higher before the fork happened.

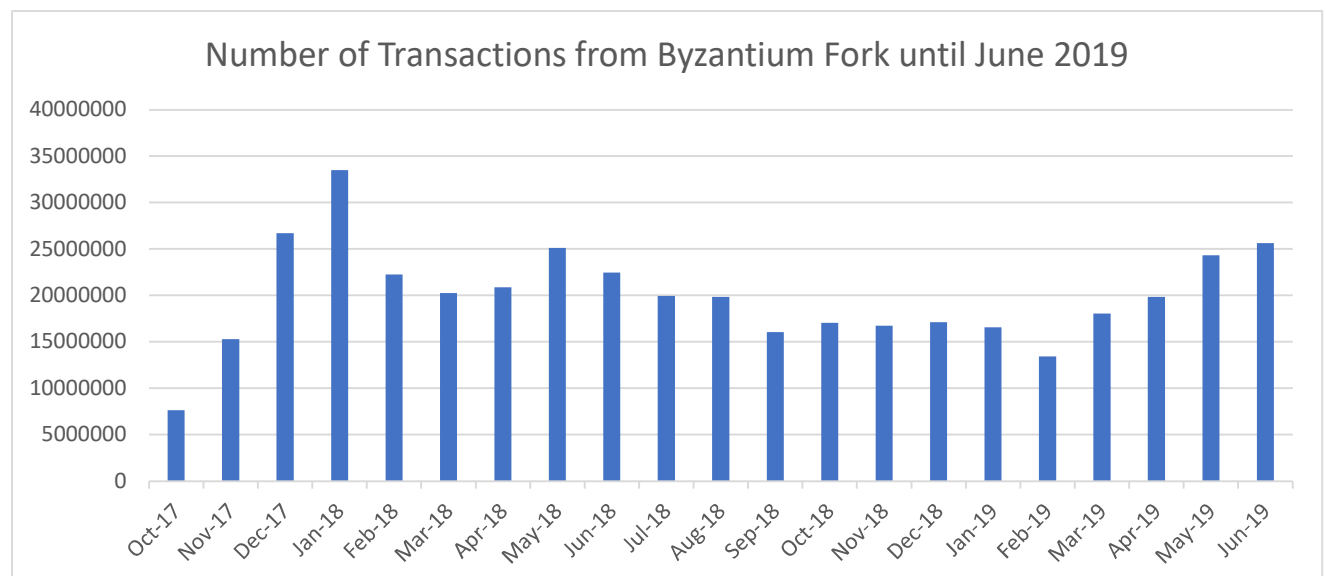
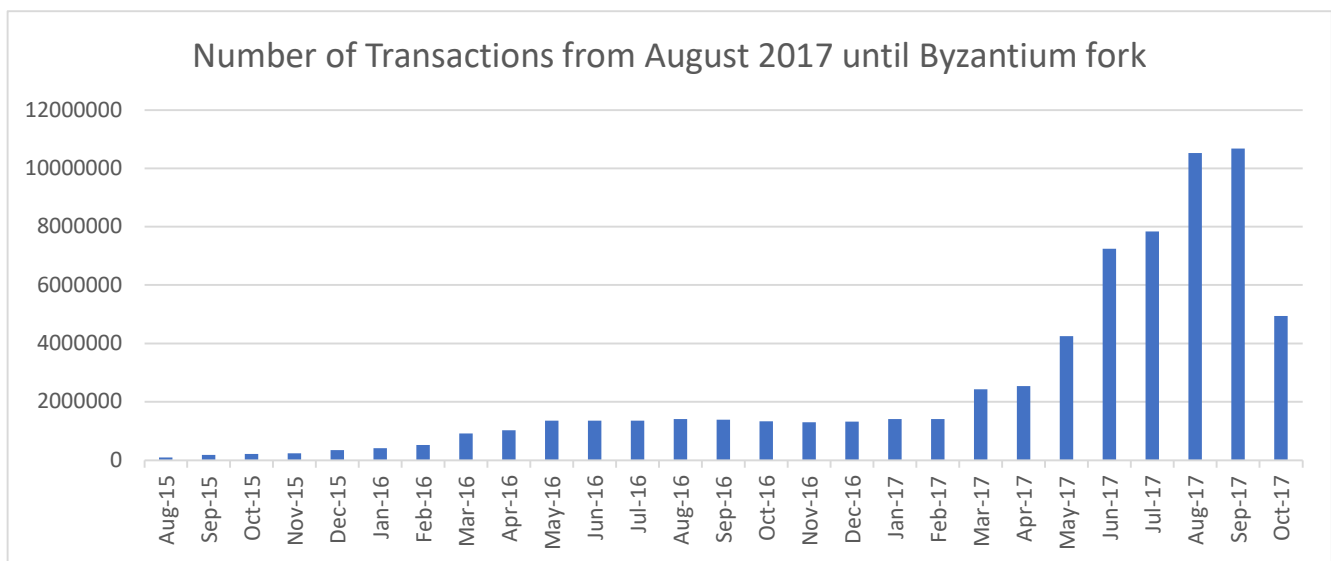
In order to see any change in general usage, the number of transactions will be taken as it can tell us how many times a transaction has occurred for a given time. The number of transactions will be worked out for both before and after the fork. The next 2 jobs run will be the same, with the difference being the focus on either before or after the fork.

The mapper will take the timestamp of the transaction from the fields and will be used to see if the transaction took place before the fork. If it is, then the transaction will get passed to the combiner. The timestamp would be formatted into a date and that would be sent as the key. The value that would be passed would be a 1 as we are getting the total number of transactions and so at the end all these 1s being passed will be grouped and aggregated into a single value.

The combiner will perform the same task as the reducer, which was to get the sum of the number of transactions for each month. The reducer would perform the very same task. They would both return the date as the key, and the total number of transactions as the value. This was similar to Part A for the number of transactions only this time we are only taking in values that were before the fork timestamp.

For the next job, we then do the same thing but this time we will take in transactions AFTER the fork. The mapper, combiner and reducer tasks stay the same with the difference of this time only taking in timestamps that are greater than the fork timestamp.

The following graphs were plotted based on their respective outputs:



We can now compare the two graphs together as we have both graphs from before and after the fork. When we look at it, we can see the opposite has actually happened with the number of transactions. The number of transactions after the fork has actually increased. This means that the general usage of Ethereum has increased.

Finally, the final task is to gather the top 10 most lucrative addresses for after the fork happened. This is so that we are able to get the highest one, because that will be the address that profited the most from the fork.

The mapper of this job would simply take all the transactions of whose timestamps are larger than the fork timestamp and send it to the combiner. If the timestamp is smaller, it will not be used.

The combiner and reducer will simply sum up the values for the associated keys. The combiner will send it to the reducer which will then perform the very same task as the combiner but upon finishing the job it will send the outputs to the mapper of the next job which will do the top 10 job.

For the next job which does the top 10, the mapper will just send the data to the combiner in the values, nothing will be sent as the key.

The combiner will get these values and then sort the tuples based on the aggregated value of the previous job. Once complete, a for loop will then output the first 10 tuples which is sorted from highest to lowest. The key sent will just be a filler "top", and value will contain the tuple with the address and aggregated value.

The reducer will do the same thing as the combiner, but when outputting the result, the address will be outputted as the key and the aggregated value will be outputted as the value.

The following results were obtained from the output:

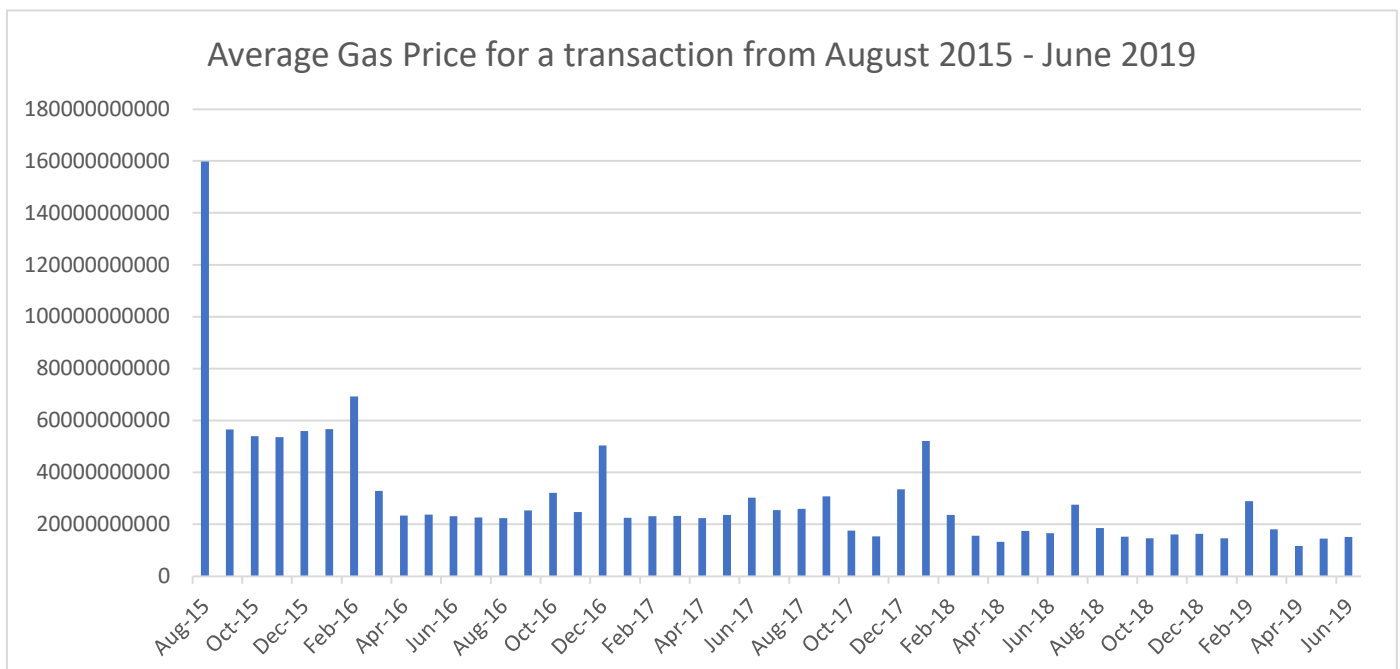
"0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be"	57415617140386781900189883
"0x876eabf441b2ee5b5b0554fd502a8e0600950cfa"	40157678878619363035574179
"0xfa52274dd61e1643d2205169732f29114bc240b3"	17469680105843451546319513
"0x6cc5f688a315f3dc28a7781717a9a798a59fda7b"	17327004263271962150196572
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"	15093309292215332077995484
"0xd551234ae421e3bcba99a0da6d736074f22192ff"	11851425263280827647019849
"0x5e032243d507c743b061ef021e2ec7fcc6d3ab89"	11829169343782923136145277
"0x0681d8db095565fe8a346fa0277bffd9c0edbbf"	11823716433166622712687133
"0x564286362092d8e7936f0549571a803b203aaced"	11492690787809090413153222
"0xf4a2eff88a408ff4c4550148151c33c93442619e"	9256195852226654400425950

We can see that after the Byzantium fork, these were the top 10 most lucrative addresses. To answer the question of which profited the most, the first address at the top profited the most following the fork as shown with the aggregated value. This means that this address gained the most money following the fork.

Gas Guzzlers

The average gas price was calculated for each month from the transactions dataset for every year and was plotted in a bar chart.

The way I calculated the average gas price was basically the same as to how Part A was performed with regards to the average value. However instead of using the values column, the column with the gas price was used with the nearly the same code as Part A. The output provided results for the graph below:



Upon the results, based on the average gas prices, it is clear that the gas price for a transaction has decreased.

In order to find out whether contracts have become complicated or not, the difficulty section in the blocks dataset will help knowing if contracts have become difficult. The higher the number in difficulty, the more complicated it is. However, we also need to know if more gas is needed or if less gas is needed. Firstly, the difficulty was calculated in the next job.

A join needs to be performed with contracts and blocks as we are looking for if contracts have become complicated or not. A repartition join is required for this. The first job of the MapReduce program will focus on joining the two datasets together. Afterwards, the average difficulty of a contract for a given time was worked out.

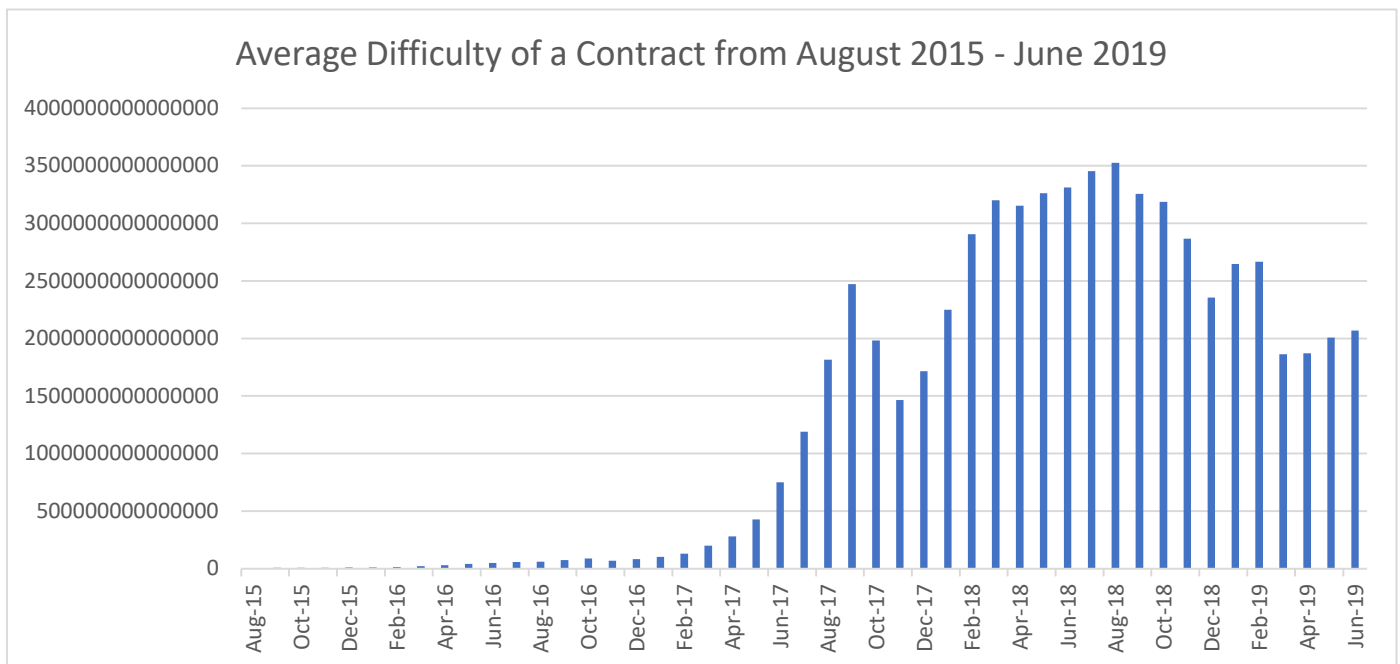
The mapper would take in the datasets of blocks and contracts. The mapper of the repartition join would assign the necessary values for the join. The block number was used to join the datasets together. The timestamp and the difficulty columns were the columns that were needed for this job. The time was converted from a timestamp into a date format. The key was the block number and the value was a tuple containing the time, the difficulty and the integer 2. At the contracts dataset, the only thing that was taken was the block number which was the key, and 3 instances of the integer 1 being placed in the tuple as the value, two of the elements are just fillers so that it can match the number of elements in the tuple. The mapper would send this to the reducer.

The reducer is where it checks if the block numbers from the two datasets match. If the output is from the contracts dataset since the last element is 1, then the flag is switched to true. If the output is from the blocks dataset, then the variables of the time and difficulty are assigned to their respective variables as the last element is 2. At the end, if the flag is true and there is a number in the difficulty variable, then the variables are outputted to the mapper of the average task. The key being sent is the time and the value will be the difficulty.

Upon the start of the average job, the mapper will send the data it receives from the reducer of the previous job to the combiner of the current job.

The combiner will get the values and work out the total number of the blocks associated with the date as well as the sum of the value. Once completed, the combiner will send the key containing the time and the value containing the total and aggregate value for the associated block respectively to the reducer, where the reducer will perform the same task but rather than sending the total and sum of the value as a tuple it will work out the average. The reducer will output the results.

After the conclusion of all the jobs, the outputs that were received were then plotted in this graph below:



We can see here that over time the difficulty of the contracts has risen. Basically, what this means is that the contracts have become more complicated to process as time has progressed.

Finally, our last task is that we have to find out whether if the amount of gas that is being sent has some sort of a correlation with this graph. We need to see whether less gas is sent for a more complicated contract or if more gas is sent.

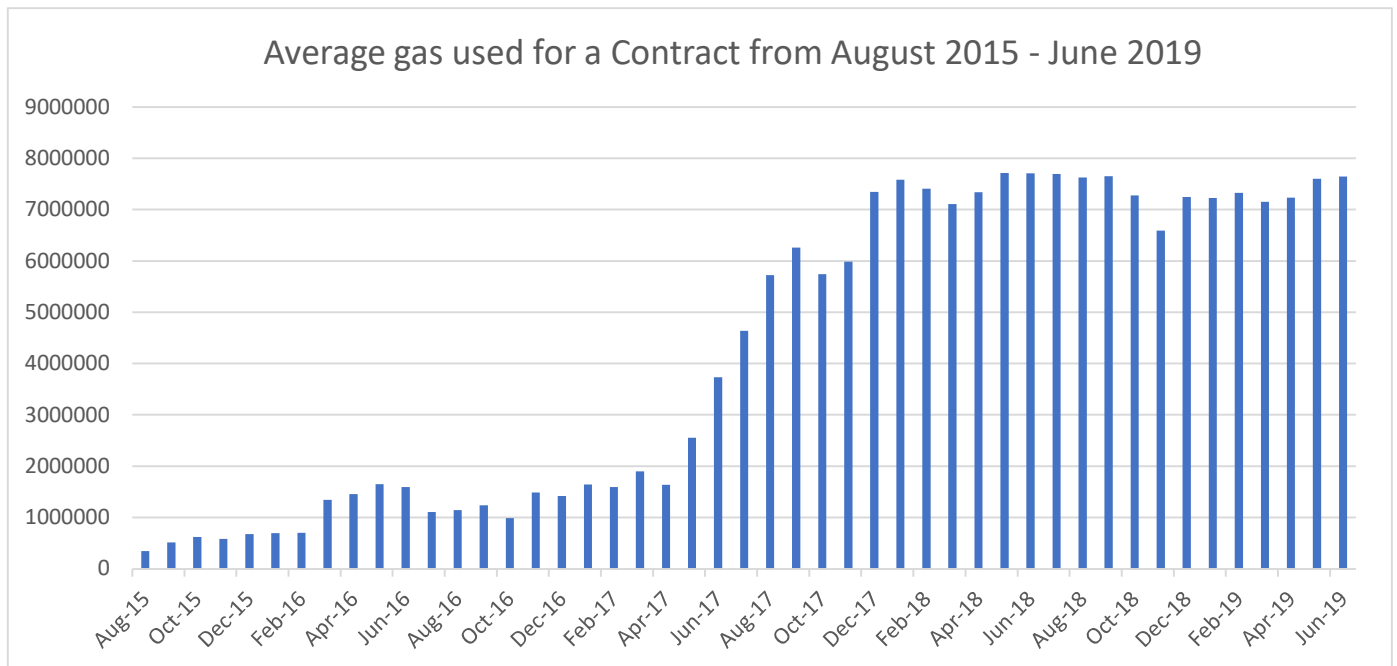
For this final task, this will basically do the same thing as the above task, however the amount of gas that was used will be focused on rather than the difficulty.

The block number will be used once again to join the datasets as we are pretty much performing the same task again. The same repartition join happens only this time we do not take the difficulty, but we take the `gas_used` section of the blocks instead. The reducer will send the outputs to the mapper of the average task.

The mapper will receive the outputs and will do the same thing as before, which is to just send the data to the combiner.

The combiner will perform the same task as before, only this time the value that is focused on is the gas amount used. Once completed, the combiner will send the output to the reducer, where the reducer will work out the average of the value at the end. The reducer after finishing the task will output the results.

The output of this task was received and plotted in the graph below:



The results from this graph show that the average gas sent has also increased over time. This means that as the difficulty has increased for a contract, the amount of gas you must send for a contract has also gone up. This means that the more difficult the contract, the more gas you must pay.

Comparative Evaluation

To compare both my MapReduce program for Part B and its Spark counterpart, the result I would use to compare the two would be the time taken to operate the exact same task. I ran the operation three times each with both programs and worked out the median value for each of the two. To make sure that my Spark results were correct, they needed to be the same answer as my actual Part B, which was done so below.

```
hg304@itl002 ~/ECS640U/cw/Part D/Part B in Spark> spark-submit partbinspark.py
21/11/29 14:32:31 WARN lineage.LineageWriter: Lineage directory /var/log/spark/1
Lineage for this application will be disabled.
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444, 84155100809965865822726776
0xfa52274dd61e1643d2205169732f29114bc240b3, 45787484483189352986478805
0x7727e5113d1d161373623e5f49fd568b4f543a9e, 45620624001350712557268573
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef, 43170356092262468919298969
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8, 27068921582019542499882877
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd, 21104195138093660050000000
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3, 15562398956802112254719409
0xbb9bc244d798123fde783fcc1c72d3bb8c189413, 11983608729202893846818681
0xabbb6bebfa05aa13e908eaa492bd7a8343760477, 11706457177940895521770404
0x341e790174e3a4d35b65fdc067b6b5634a61caea, 8379000751917755624057500
```

I first obtained both the transactions and contracts datasets. I then filtered the two datasets so that malformed lines were removed using the python functions that I made that would take care of this situation, I have two functions, one for transactions and one for contracts as they have different number of fields in their lines. 7 for transactions and 5 for contracts.

Once filtering was complete the mapping tasks for the two were performed, the transactions would be mapped with the address as the key and its value as the value, and the contracts would have the address as the key and the integer 0 as the value. The value 0 will be needed for a task ahead.

A reduce function is performed afterwards with the transaction tuples and the values will be grouped by its respective key and once grouped the values are summed for each key into one value. This means that each key will have the sum of all the values associated with it.

Because the reduce operation is an action in Spark, we have to use `persist()` in order to keep it in memory, as RDDs can be removed once an action has been executed. If not used, it can cause the execution time to be longer as the RDD may have to be computed again.

We then join this tuple with our contracts dataset. The transactions with a contract will have an extra value added in the values element, which will be the 0. This is where the 0 is used. Any value that doesn't have a 0 will be removed as it signifies that the transaction is not part of a contract and thus will be removed in the next transformation which is `filter()`.

Finally the top 10 operations takes place with `takeOrdered()` which sorts the values in order. In our case we want it to go from highest to lowest, and we want only the top 10 as we signify in the operation. Once completed we then finally do a for loop where these values are then outputted from highest to lowest, to the machine. Our results are correct because we got the same results as our part B with MapReduce.

To analyse between the two, I ran these two programs three times and calculated the time taken for each program to finish in order to see which performs this tedious task much faster. I would then get the average time from these answers.

The times taken when running the MapReduce program were:

Attempt 1: 36 min 20 sec (2180 seconds)

Attempt 2: 35 min 27 sec (2127 seconds)

Attempt 3: 34 min 59 sec (2099 seconds)

The times taken when running the Spark program were:

Attempt 1: 3 min 10 sec (190 seconds)

Attempt 2: 2 min 51 sec (171 seconds)

Attempt 3: 2 min 43 sec (163 seconds)

It is evidently clear that running the same task on Spark is significantly faster than on MapReduce. The average time for MapReduce was $(2180 + 2127 + 2099) / 3 = 2135$ seconds, which is 35 mins 35 seconds. The average for Spark was $(190 + 171 + 163) / 3 = 175$ seconds, which is 2 mins 55 seconds.

The reason as to why Spark is much faster than MapReduce is because Spark processes data in memory. Unlike MapReduce, Spark uses in memory processing meaning that it will perform the actions in the memory rather than the disk drive. Spark will also keep this data in memory for any further steps speeding up tasks further, unlike MapReduce which processes on the disk drive. Processing data with Spark is up to 100 times faster when comparing it with MapReduce.

As a result, the best framework to use for a task like this is Spark, if we use MapReduce then due to the data being processed on disk we will have to wait on average half an hour for the job to complete. Additionally, our task on MapReduce ran with two jobs whereas with Spark it did it in one. With Spark, we can do this entire task in memory which is faster giving an average time of just under 3 minutes.