Sammi Pang 190153630

Big Data Ethereum Analysis Coursework Report

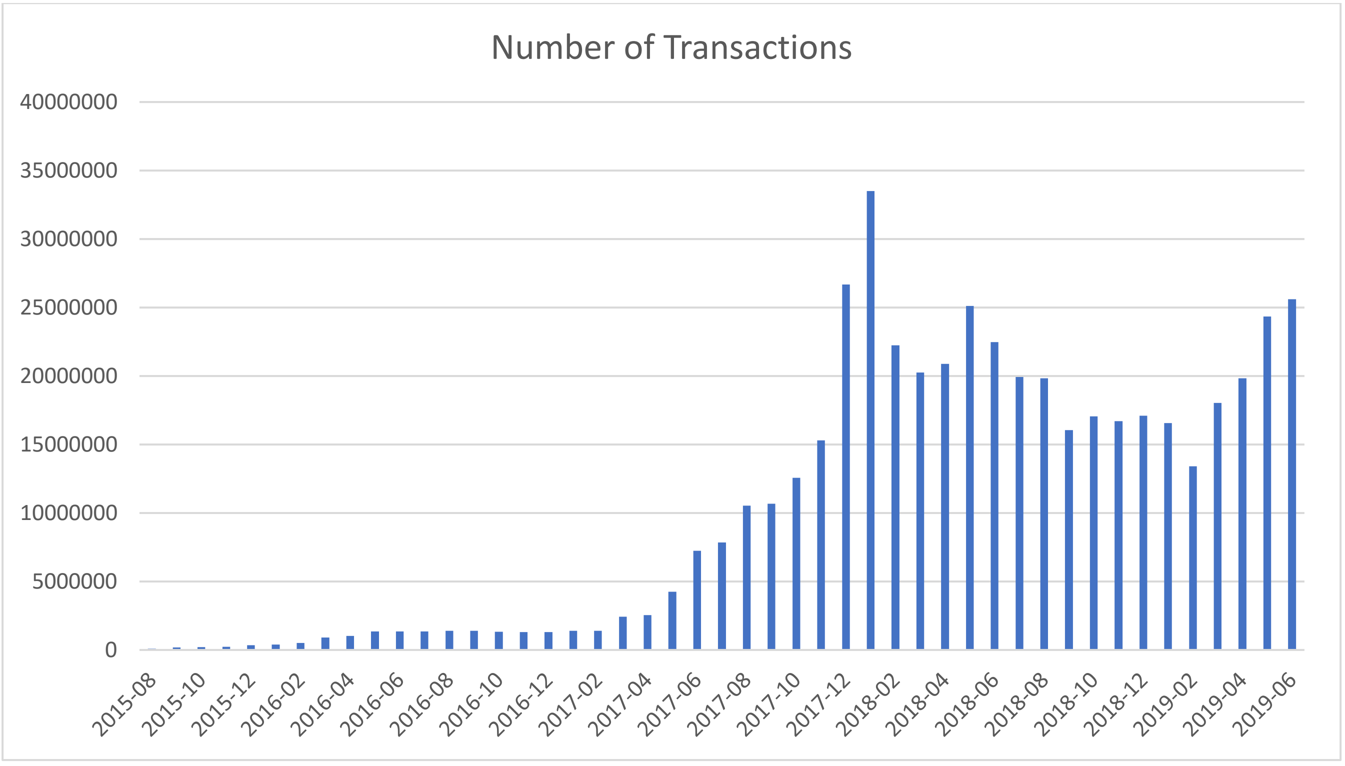
Part A:

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

In Part A I calculated the number of transactions occurring every month using MapReduce in my code numTrans.py. For my input I used the transactions table from data/Ethereum/transactions. In my mapper I split the fields using a comma I read the timestamp from the dataset and convert this into a date format of Year-Month. I do this using the strftime and gmtime function. I then yield the date as the key and the value 1, this is because everytime I am reading a month in from the mapper I am also reading a transactions so I yield 1 to keep track of each transaction I am passing, which I will later accumulate.

For my reducer I take the inputs of the key which is the date and the value which is the count for the transactions. I sum up the count for transactions to get the total of the transactions for that month since my reducer will reduce the data by the date and then yield the final date and total of transactions.

Using Excel I then plotted the results in a bar chart where you can see below as the months pass the number of transactions increase.

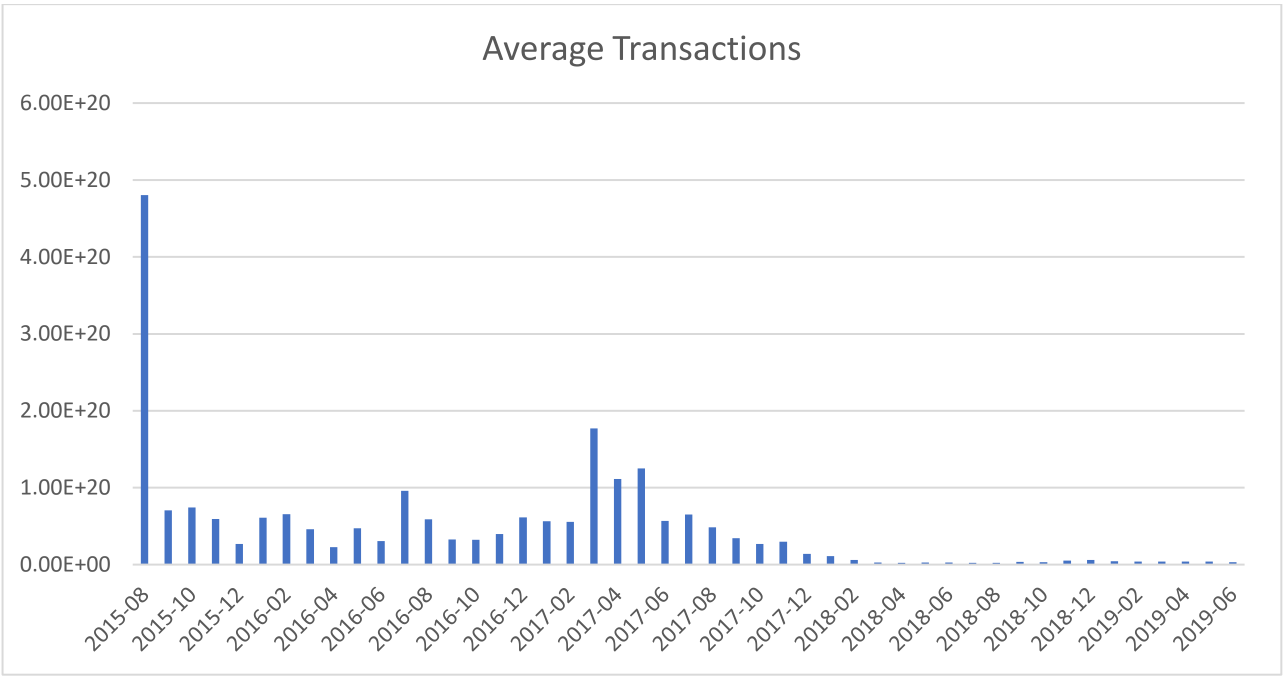


Create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

In this part of Part A I calculated the average value of transactions in each month. I used a similar program to my numTrans.py program but instead in my avgTrans.py I take the value of Wei in a transaction from the dataset data/Ethereum/transactions. I then yield the month and the amount of Wei passed in that transaction for that month in my mapper.

For my reducer I have a counter where I use a for loop and increment this counter, to keep track of the number of transactions I have passed. In this for loop I add up the total amount of Wei passed in that month. I then proceed to divide the sum of Wei by the amount of transactions passed in that month, thus leading me to yield the month, as the key, and the average of Wei passed in that month, as the value.

From Excel I plotted a graph and you can see that more Wei was passed in transactions during the months in 2015-2017 and slowly started declining towards the later years.



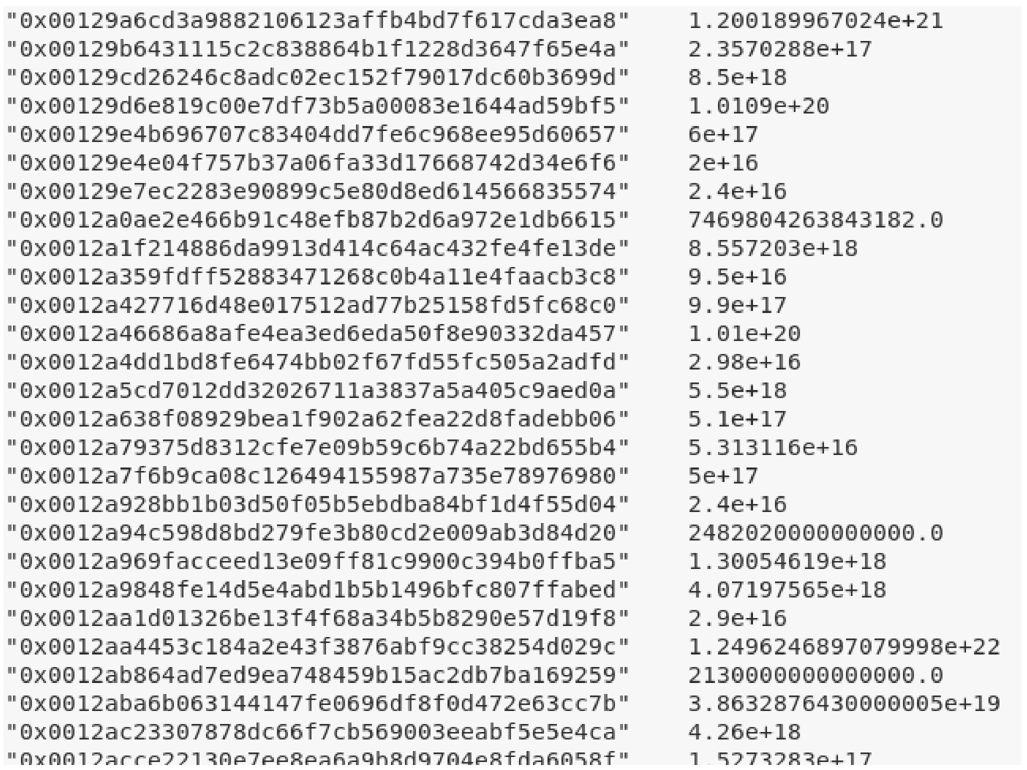
Part B

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.

In Job 1 of Part B, topJob1.py, I use MapReduce again and aggregate from the transactions table, data/Ethereum/transactions. I proceed to split the fields by a comma and I get the to_address and the value of Wei transferred. I then yield these values using the address as my key and the value of Wei as my value.

In my reducer I sum up all the Wei transferred for the specific address being reduced and then yield that address as the key with the total amount of Wei transferred as the value. Thus, producing the result below.



```
"0x00129a6cd3a9882106123affb4bd7f617cda3ea8"    1.200189967024e+21
"0x00129b6431115c2c838864b1f1228d3647f65e4a"    2.3570288e+17
"0x00129cd26246c8adc02ec152f79017dc60b3699d"    8.5e+18
"0x00129d6e819c00e7df73b5a00083e1644ad59bf5"    1.0109e+20
"0x00129e4b696707c83404dd7fe6c968ee95d60657"    6e+17
"0x00129e4e04f757b37a06fa33d17668742d34e6f6"    2e+16
"0x00129e7ec2283e90899c5e80d8ed614566835574"    2.4e+16
"0x0012a0ae2e466b91c48efb87b2d6a972e1db6615"    7469804263843182.0
"0x0012a1f214886da9913d414c64ac432fe4fe13de"    8.557203e+18
"0x0012a359fdff52883471268c0b4a11e4faacb3c8"    9.5e+16
"0x0012a427716d48e017512ad77b25158fd5fc68c0"    9.9e+17
"0x0012a46686a8afe4ea3ed6eda50f8e90332da457"    1.01e+20
"0x0012a4dd1bd8fe6474bb02f67fd55fc505a2adfd"    2.98e+16
"0x0012a5cd7012dd32026711a3837a5a405c9aed0a"    5.5e+18
"0x0012a638f08929bea1f902a62fea22d8fadebb06"    5.1e+17
"0x0012a79375d8312cfe7e09b59c6b74a22bd655b4"    5.313116e+16
"0x0012a7f6b9ca08c126494155987a735e78976980"    5e+17
"0x0012a928bb1b03d50f05b5ebdba84bf1d4f55d04"    2.4e+16
"0x0012a94c598d8bd279fe3b80cd2e009ab3d84d20"    2482020000000000.0
"0x0012a969facceed13e09ff81c9900c394b0ffba5"    1.30054619e+18
"0x0012a9848fe14d5e4abd1b5b1496bfc807ffabed"    4.07197565e+18
"0x0012aa1d01326be13f4f68a34b5b8290e57d19f8"    2.9e+16
"0x0012aa4453c184a2e43f3876abf9cc38254d029c"    1.2496246897079998e+22
"0x0012ab864ad7ed9ea748459b15ac2db7ba169259"    2130000000000000.0
"0x0012aba6b063144147fe0696df8f0d472e63cc7b"    3.8632876430000005e+19
"0x0012ac23307878dc66f7cb569003eeabf5e5e4ca"    4.26e+18
"0x0012acce22130e7ee8ea6a9b8d9704e8fda6058f"    1.5273283e+17
```

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

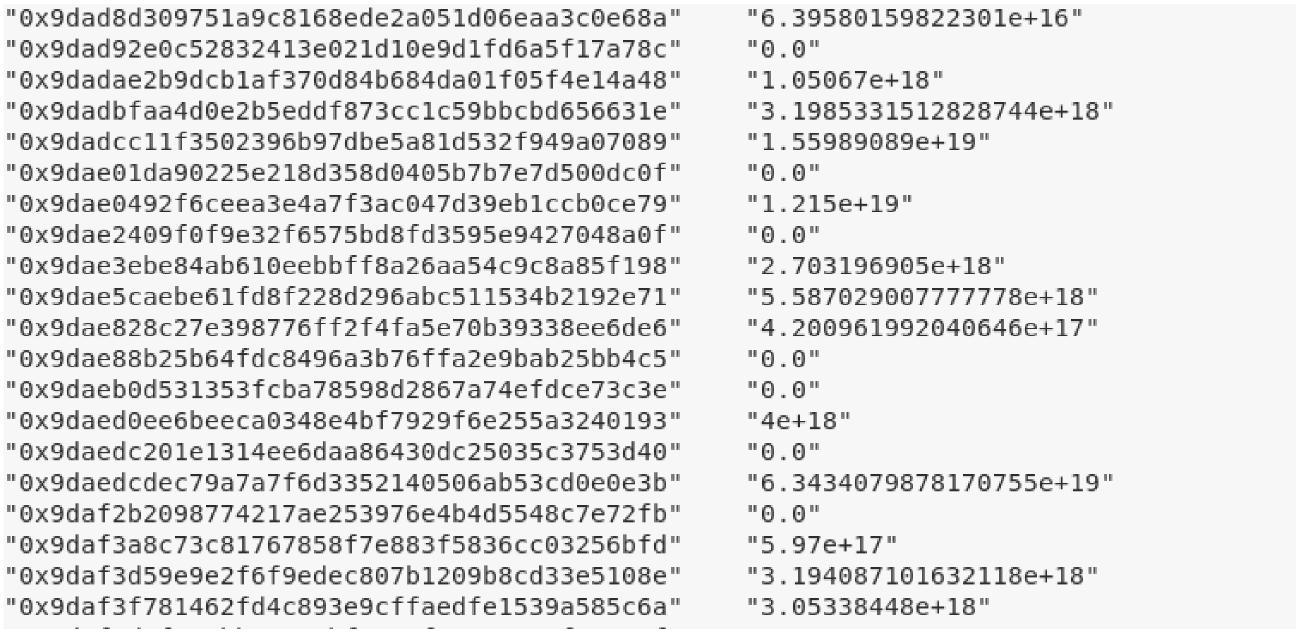
Secondly, in the reducer, if the address for a given aggregate from Job 1 was not present within contracts this should be filtered out as it is a user address and not a smart contract.

To note when running this program the contracts dataset should be the first input then followed by the aggregate dataset made in Job 1.

Here I perform a repartition join through my mapper and reducer in my code top2.py. I do an if statement to make sure I am reading the right value by checking the number of fields. I read the txt file, I made in my Job 1 with the aggregated results of address and value of wei. I use an any split (tab) to split the fields making the address the join key and the value of wei the value. I make a compound value with another value of integer 2, which will help later with identifying it in the reducer. I also read the contracts dataset data/Ethereum/contracts and split using a comma. I make the join key the address of the contract and I have a compound value, where I pass the unique integer identifier 2 and a value of None.

In my reducer I make an if statement to help identify what key, values I am reading from the mapper using the integer values 1 and 2 I used earlier in my mapper. I make a list to store all the address of the contracts to use later. When reading the key, values from my aggregate dataset I check whether the address from that dataset is in the contracts address list I made earlier. If yes, then it will yield this address along with the join value (total amount of Wei passed), otherwise it will filter out the address and skip it as this address does not exist in contracts.

This is the part of the output of my results from the job.



```
"0x9dad8d309751a9c8168ede2a051d06eaa3c0e68a"    "6.39580159822301e+16"
"0x9dad92e0c52832413e021d10e9d1fd6a5f17a78c"    "0.0"
"0x9dadae2b9dcb1af370d84b684da01f05f4e14a48"    "1.05067e+18"
"0x9dadbfaa4d0e2b5eddf873cc1c59bbcbd656631e"    "3.1985331512828744e+18"
"0x9dadcc11f3502396b97dbe5a81d532f949a07089"    "1.55989089e+19"
"0x9dae01da90225e218d358d0405b7b7e7d500dc0f"    "0.0"
"0x9dae0492f6ceea3e4a7f3ac047d39eb1ccb0ce79"    "1.215e+19"
"0x9dae2409f0f9e32f6575bd8fd3595e9427048a0f"    "0.0"
"0x9dae3ebe84ab610eebbff8a26aa54c9c8a85f198"    "2.703196905e+18"
"0x9dae5caebe61fd8f228d296abc511534b2192e71"    "5.587029007777778e+18"
"0x9dae828c27e398776ff2f4fa5e70b39338ee6de6"    "4.200961992040646e+17"
"0x9dae88b25b64fdc8496a3b76ffa2e9bab25bb4c5"    "0.0"
"0x9daeb0d531353fcba78598d2867a74efdce73c3e"    "0.0"
"0x9daed0ee6beeca0348e4bf7929f6e255a3240193"    "4e+18"
"0x9daedc201e1314ee6daa86430dc25035c3753d40"    "0.0"
"0x9daedcdec79a7a7f6d3352140506ab53cd0e0e3b"    "6.3434079878170755e+19"
"0x9daf2b2098774217ae253976e4b4d5548c7e72fb"    "0.0"
"0x9daf3a8c73c81767858f7e883f5836cc03256bfd"    "5.97e+17"
"0x9daf3d59e9e2f6f9edec807b1209b8cd33e5108e"    "3.194087101632118e+18"
"0x9daf3f781462fd4c893e9cffaedfe1539a585c6a"    "3.05338448e+18"
```

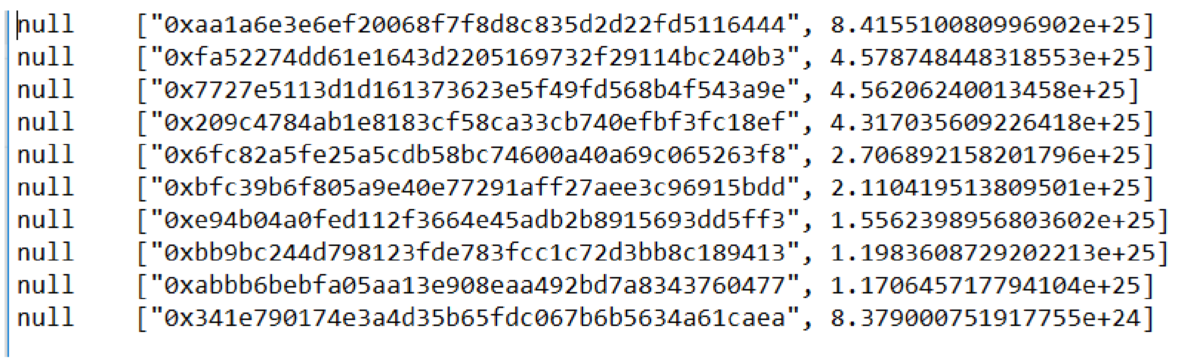
JOB 3 - TOP TEN

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.

For this step I want to get the top 10 values of the addresses with the highest total of Wei transferred, from my code topJob3.py. In my mapper I read the file I outputted from Job 2 and I yield None for the key and make a compound value made up of the address and the total Wei.

In my reducer I sort the list of values using the sorted function and lambda. I make sure to sort by the total amount of Wei. I then do a for loop, which repeats 10 times, where I yield the null key and the sorted values, which are sorted from largest to smallest.

Leading to the following output



```
null	["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", 8.415510080996902e+25]
null	["0xfa52274dd61e1643d2205169732f29114bc240b3", 4.578748448318553e+25]
null	["0x7727e5113d1d161373623e5f49fd568b4f543a9e", 4.56206240013458e+25]
null	["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", 4.317035609226418e+25]
null	["0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8", 2.706892158201796e+25]
null	["0xbfc39b6f805a9e40e77291aff27aee3c96915bdd", 2.110419513809501e+25]
null	["0xe94b04a0fed112f3664e45adb2b8915693dd5ff3", 1.5562398956803602e+25]
null	["0xbb9bc244d798123fde783fcc1c72d3bb8c189413", 1.1983608729202213e+25]
null	["0xabbb6bebfa05aa13e908eaa492bd7a8343760477", 1.170645717794104e+25]
null	["0x341e790174e3a4d35b65fdc067b6b5634a61caea", 8.379000751917755e+24]
```

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

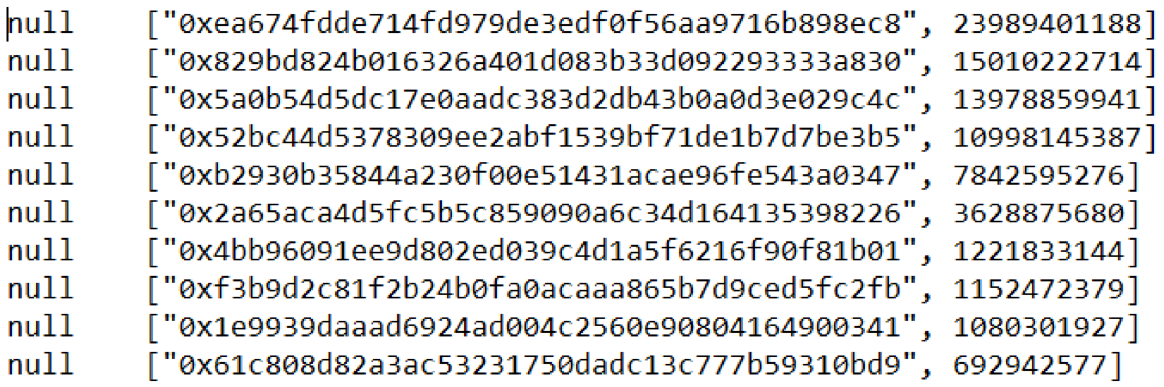
Evaluate the top 10 miners by the size of the blocks mined. This is simpler as it does not require a join. You will first have to aggregate blocks to see how much each miner has been involved in. You will want to aggregate size for addresses in the miner field. This will be similar to the wordcount that 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.

This part is essentially similar to part B but without the repartition join. I use MRStep since I will be using 2 mappers and reducers. For my input I am taking in values from the blocks dataset data/Ethereum/blocks. In my first mapper, mapper_block, I read the dataset splitting the fields by a comma. I yield the miner addresses as the key and for my value I use the size of the block.

In my first reducer, reuducer_sum, I removed duplicate miners by reducing by the key which is miner address. I then add up the total size of the blocks for that miner, yielding the miner as the key and the total size of blocks for the miner as the value.

In my next mapper, mapper_none, since I want to calculate top 10 I yield None as my key and for my value I yield the miner and the total size of blocks for the miner.

In my next reducer, reducer_sort, this is where I sort the list of values using the sorted and lambda function by the total size of blocks for the miner. I loop 10 times yielding the null key and the sorted values, which are sorted by the biggest total block size for the miner. Providing me with the results below.



```
null    ["0xea674fdde714fd979de3edf0f56aa9716b898ec8", 23989401188]
null    ["0x829bd824b016326a401d083b33d092293333a830", 15010222714]
null    ["0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c", 13978859941]
null    ["0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5", 10998145387]
null    ["0xb2930b35844a230f00e51431acae96fe543a0347", 7842595276]
null    ["0x2a65aca4d5fc5b5c859090a6c34d164135398226", 3628875680]
null    ["0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01", 1221833144]
null    ["0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb", 1152472379]
null    ["0x1e9939daaad6924ad004c2560e90804164900341", 1080301927]
null    ["0x61c808d82a3ac53231750dadc13c777b59310bd9", 692942577]
```

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 graphy analysis. Below are some suggested ideas for analysis which could be undertaken, along with an expected grade for completing it to a *good* standard. 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

1. **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)**

I first converted the json file, scams.json, to a csv using my code jsonToCSV.py. I converted the json to csv printing only the results I wanted, which was the addresses associated with the scam, unique ID for the reported scam, the category of scam, and the status of the scam (offline/active).

In my code scam1.1.py I made an aggregate result from the data/Ethereum/transactions dataset. I yielded a compound key of the from_address and the month (using timestamp), and a value of the value of Wei transferred for each transaction in my mapper. In my reducer I them summed the amount of transaction for that specific address and month. Yielding the address and month as the key and the total sum of Wei transferred as the value.

From here I used my code scam1.py. I read the csv version of scams.json and my aggregated dataset from scam1.1.py in my mapper. For the csv file I yield the address as the join key and a compound value of a unique file identifier of 2, the scam status and the scam category. For my aggregated dataset I read in the values address as my join key, a compound value with a unique file identifier of 1, total Wei transferred, and the month.

In my reducer I make a dictionary to map the status and category from the csv file with each other, since I am yielding the address with the month in the reducer I want to keep track of which address has which status and category, storing them in a dictionary. I also store the month of each address and value of Wei in separate variables when reading the aggregated dataset. I finally go through every value in the dictionary and yield with a compound key of address and month. And a compound value of total value of Wei, status, and the category of the scam.

Unfortunately due to disk quota on the HDFS I was unable to run this job and continue with my scams analysis. However, should I have continued I would have then done a top 10 of the results from this job and sorted them by the total about of Wei, thus yielding the address and month with the high amount of Wei transferred. I would then check the category of the scam, thus answering the question “what is the most lucrative form of scam?” . After this I could have done top 10 again but sorting by the date instead plotting my results on a graph with time against value of Wei transferred to see if the amount of Wei accumulated changes over time. Judging by the results I could manually see the type of scam and their status to see which scam yielded the most Wei in this set of 10 and whether these scams were active or offline majority of the time.

Contract Types

1. **Identifying Contract Types The identification and classification of smart contracts can help us to better understand the behavior of smart contracts and figure out vulnerabilities, such as confirming fraud contracts. By identifying features of different contracts such as number of transactions, number of uniques outflow addresses, Ether balance and others we can identify different types of contracts. How many different types can you identify? What is the most popular type of contract? More information can be found at https://www.sciencedirect.com/science/article/pii/S0306457320309547#bib0019. Please note that you are not required to use the types defined in this paper. Partial marks will be awarded for feature extraction and analysis. (25/50)**

Here I decided to use some of my knowledge from my data mining module, however I was unable to completely finish the task due to my lack of knowledge.

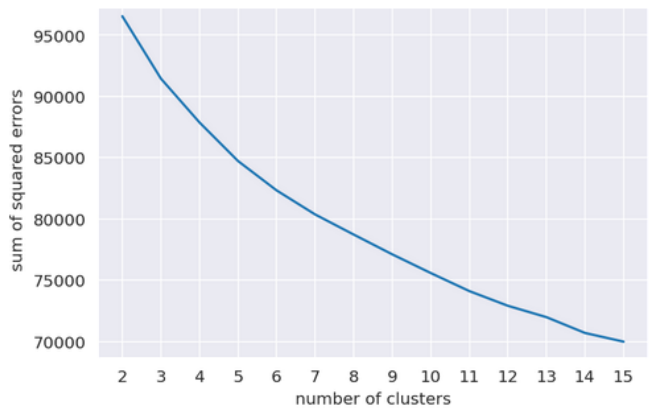
I decided to make a repartition join with contracts and transactions dataset data/Ethereum/transactions, dataset data/Ethereum/transactions in my code contracts.py. For transactions I yield the values to_address as the join key and a compound value of a unique file identifier of 2, amount of Wei transferred, and the gas price. For contracts I yield the address as the join key, with a compound value of a unique file identifier,1, and None.

Similar to my part B I then append all the address in contracts into a list. I then look up the address from my transactions file from the mapper and if this address is in the list I yield the address as the key, as well as the amount of Wei transferred and gas price for that address. Otherwise I filter out these addresses and skip to the next address.

From here I wanted to use the sklearn library for Python to help classify the data. In order to do this I need to pre-process my data where I would have needed `from sklearn.pipeline import make_pipeline` to create a pipeline object to fit, predict and score my data in my aggregated dataset. This is to ensure the test accuracy, so that it won't overestimate the accuracy since we don't know whether our dataset has outliers and how many outliers it includes.

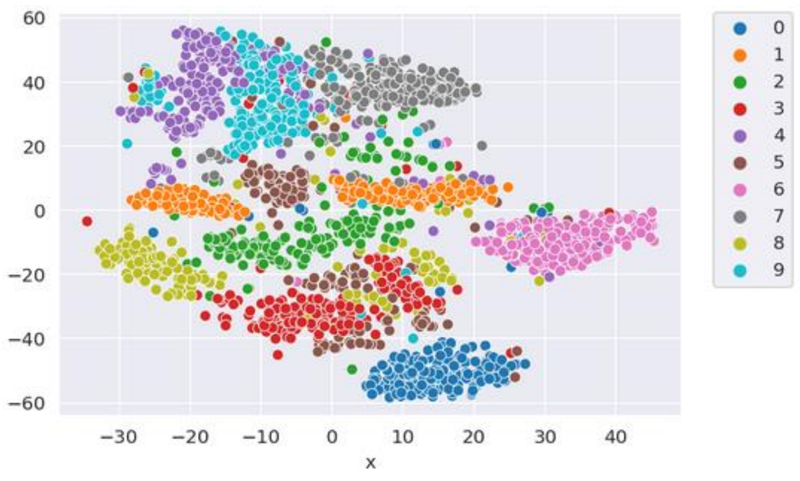
I believe I could then make a confusion matrix which will show me the number of target classes there are, using the actual target values and the ones predicted by the machine learning model(K-nearest neighbours). Using `KMeans` I could then fit my training data and make a prediction of each attribute to a class/cluster. After the prediction is computed we can check the sum of squared errors and create a regression model, this will show whether our model used to predict is good or bad depending on how low or high the sum of squares error is.

Sammi Pang 190153630



Example of a good regression model, from sum of squared errors.

I can use `TSNE` to make a projection using the t-stochastic neighbour embedding algorithm. Although this project is not a reliable representation of the dataset we can take a rough guess at which contract was the most popular type. Similar in the example I provided:

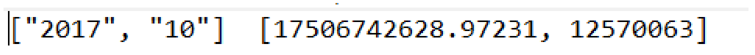


MISCELLANEOUS ANALYSIS

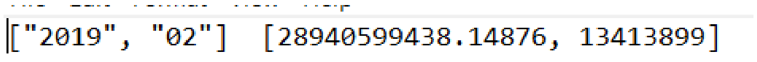
1. **Fork the Chain: There have been several forks of Ethereum in the past. Identify one or more of these and see what effect it had on price and general usage. For example, did a price surge/plummet occur and who profited most from this? (10/50)**

From looking at https://ethereum.org/en/history/#2019 I decided to analyse the forks Byzantium (Oct-16-2017) and Constantinople (Feb-28-2019). I decided to analyse the specific months these forks were made, October 2017 and February 2019. To do this I read the input from the data/Ethereum/transactions dataset. For Byzantium, refer to forkYield2017.py, I yield in my mapper the address of the receiver as the key and for the compound value I yield the gas price, counter of 1, year and month. In my reducer I add up the count, which is the number of transactions in that month, the total gas price. I calculate the average of the gas price for the address by doing the total gas price divided by the number of transactions for that address. I then yield to reduce duplicate addresses as the key and for my compound value I have the average gas price, total number of transactions, and the year and month. I do the same for Constantinople except I change the date to February 2019 for the date I am yielding, in my code forkYield2019.py.

After this I use the same inputs, however instead of yielding address I yield the date in my mapper as the key and a compound value of the gas price and count of 1 for each transaction. In my reducer I add up the total gas price and the number of transactions and compute the average by dividing sum of gas price and number of transactions. I then yield the date and the average of gas price and the number of transactions. This can be seen in my code forkUsage2017.py, and forkUsage2019.py.
With my results below this is the gas price average for that year of that month. You can see that Byzantium had 12570063 transactions and Constantinople has 131413899 transactions. You can see the correlation that the more transactions you have the higher the gas price between the two forks.



```
["2017", "10"]  [17506742628.97231, 12570063]
```

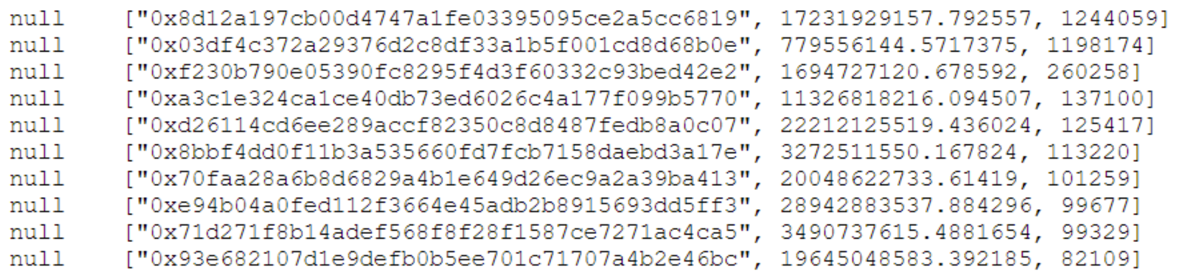


```
["2019", "02"]  [28940599438.14876, 13413899]
```

I get the top 10 for the outputs I got from my forkYield2017.py and forkYield2019.py. I do this in ascending order sorting by number of transactions in my topTen.py code. This is because I wanted to check which user had the most transactions as the address I used is the receiver's address. Meaning this receiver has had the most transactions with yield of Wei through multiple transactions in this fork for this month.

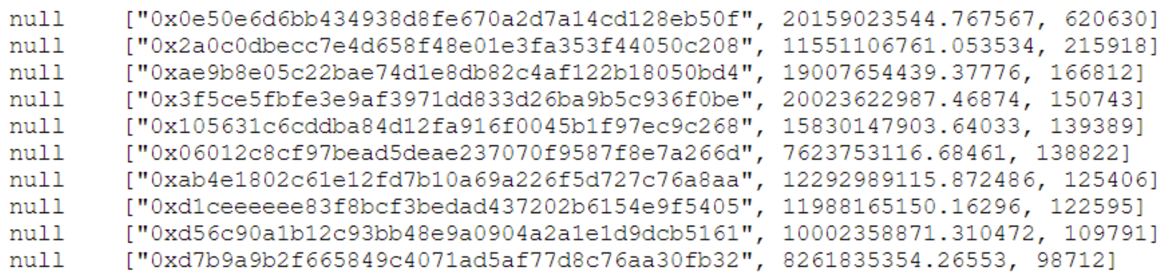
For who profited the most you can see the address of the receiver at the top of each top 10 screenshot, showing how many transactions they received in that month. Giving the assumption that they most likely yielded the most Wei profiting from the fork the most during this month and year time period.

2017



```
null    ["0x8d12a197cb00d4747a1fe03395095ce2a5cc6819", 17231929157.792557, 1244059]
null    ["0x03df4c372a29376d2c8df33a1b5f001cd8d68b0e", 779556144.5717375, 1198174]
null    ["0xf230b790e05390fc8295f4d3f60332c93bed42e2", 1694727120.678592, 260258]
null    ["0xa3c1e324ca1ce40db73ed6026c4a177f099b5770", 11326818216.094507, 137100]
null    ["0xd26114cd6ee289accf82350c8d8487fedb8a0c07", 22212125519.436024, 125417]
null    ["0x8bbf4dd0f11b3a535660fd7fcb7158daebd3a17e", 3272511550.167824, 113220]
null    ["0x70faa28a6b8d6829a4b1e649d26ec9a2a39ba413", 20048622733.61419, 101259]
null    ["0xe94b04a0fed112f3664e45adb2b8915693dd5ff3", 28942883537.884296, 99677]
null    ["0x71d271f8b14adef568f8f28f1587ce7271ac4ca5", 3490737615.4881654, 99329]
null    ["0x93e682107d1e9defb0b5ee701c71707a4b2e46bc", 19645048583.392185, 82109]
```

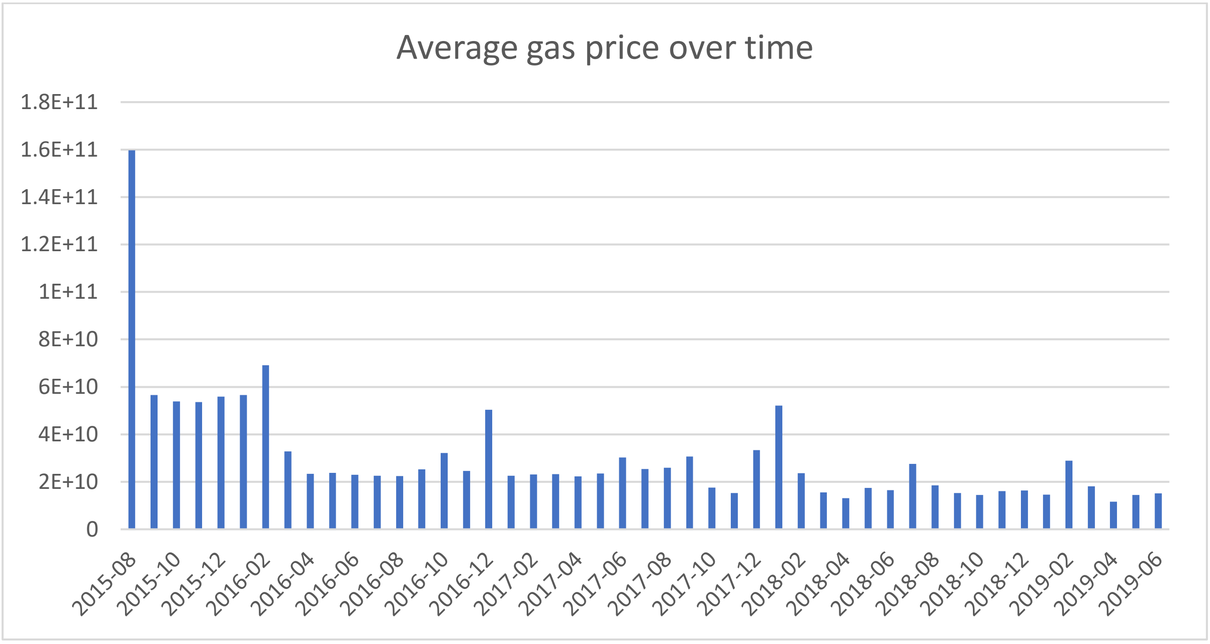
2019



```
null    ["0x0e50e6d6bb434938d8fe670a2d7a14cd128eb50f", 20159023544.767567, 620630]
null    ["0x2a0c0dbecc7e4d658f48e01e3fa353f44050c208", 11551106761.053534, 215918]
null    ["0xae9b8e05c22bae74d1e8db82c4af122b18050bd4", 19007654439.37776, 166812]
null    ["0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be", 20023622987.46874, 150743]
null    ["0x105631c6cddba84d12fa916f0045b1f97ec9c268", 15830147903.64033, 139389]
null    ["0x06012c8cf97bead5deae237070f9587f8e7a266d", 7623753116.68461, 138822]
null    ["0xab4e1802c61e12fd7b10a69a226f5d727c76a8aa", 12292989115.872486, 125406]
null    ["0xd1ceeeeee83f8bcf3bedad437202b6154e9f5405", 11988165150.16296, 122595]
null    ["0xd56c90a1b12c93bb48e9a0904a2a1e1d9dcb5161", 10002358871.310472, 109791]
null    ["0xd7b9a9b2f665849c4071ad5af77d8c76aa30fb32", 8261835354.26553, 98712]
```

2. **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)**

For the question "How has gas changed over time?". I plotted a graph with the average gas price for each month which I aggregated from the Ethereum dataset. I used inputs from the transactions dataset data/Ethereum/transactions. Here in the graph You can see the decrease of gas prices as the months pass by. Showing that gas prices have gotten cheaper in the later years.



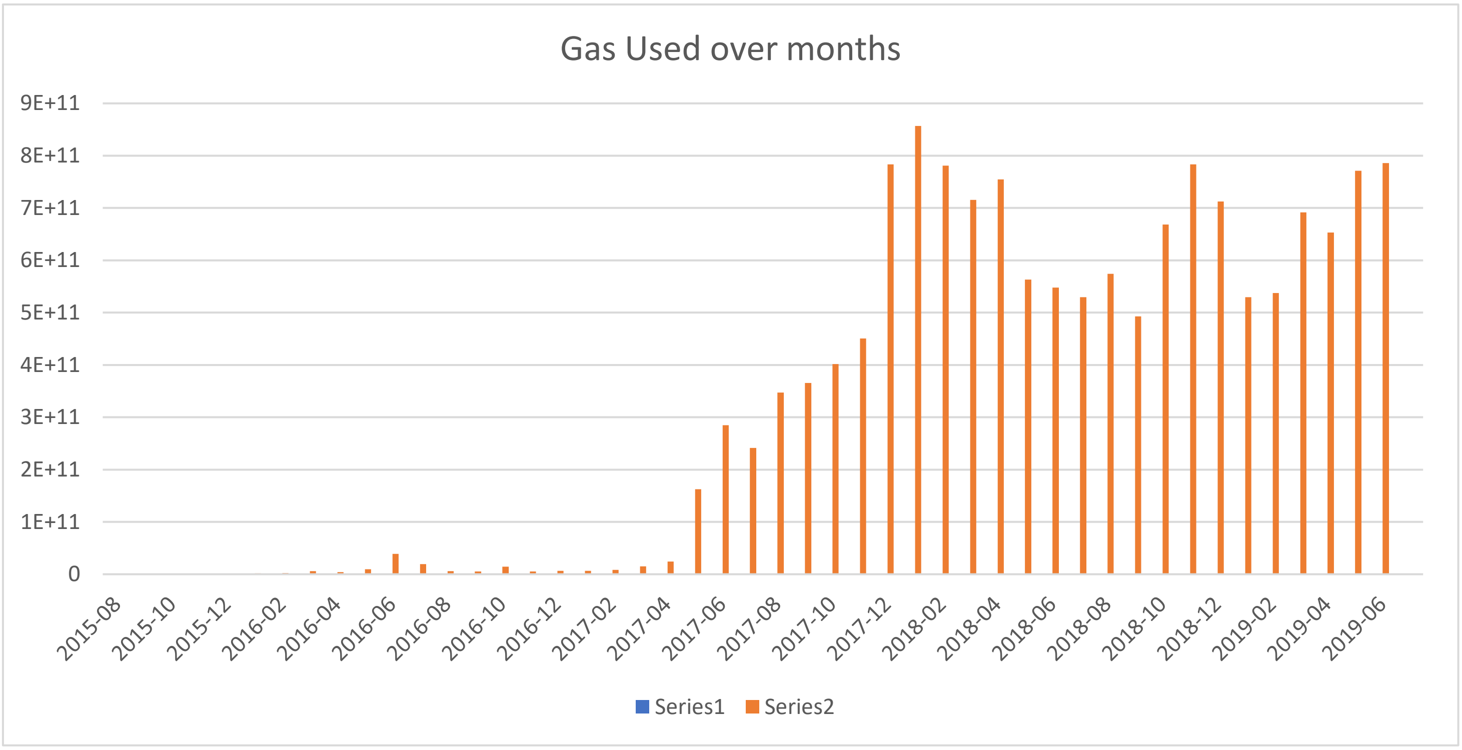
For my explanation for how I got this result refer to the code in gasAverage.py. In my mapper I read the transactions dataset, splitting the fields by a comma. I took the month and year and converted them using the timestamp and I took the gas Price from the dataset. I proceed to yield using the month as the key and making a compound value of the gas price and a count of 1 for each transaction I pass.

I make a combiner in hopes to reduce the time it takes MapReducing on the cluster, where I add up all the gas prices and the amount of transactions for that month of the year. I then yield the same values as before with my month as the key and a summed up gas price and count of transactions as the compound key.

In my reducer this is where I calculate the average and reduce the duplicate months. I add up all the gas prices for the month and the amount of transactions passed in that month. I then get the average gas price by dividing the total gas price by the total amount of transactions passed for that month. I finally yield the month as the key and the average of gas prices as the transaction.

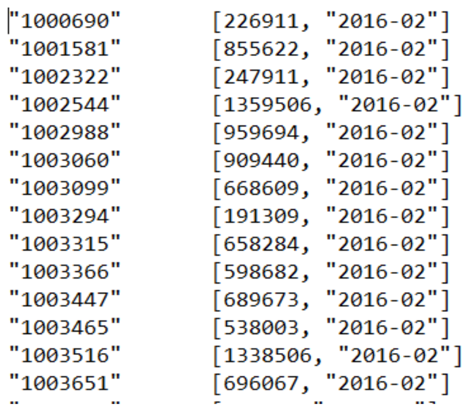
For the second question "Have contracts become more complicated, requiring more gas, or less so?" I plotted a graph of the total amount of gas used for each month, using data aggregated from the Ethereum dataset. I used inputs from the contracts and blocks table data/Ethereum/contracts data/Ethereum/blocks to gain this data for the blocks related to contracts. As you can see from the graph the amount of gas used has increased starting from 2017 onwards. This has made contracts more complicated in the sense that there is more gas being used in blocks, making these contract blocks more costly to perform transactions on the network.

https://www.investopedia.com/terms/g/gas-ethereum.asp#:~:text=On%20the%20Ethereum%20blockchain%2C%20gas,a%20transaction%20on%20the%20network.&text=The%20value%20of%20gas%20for,layer%20of%20the%20Ethereum%20platform.



As to how I achieved these results. I first had to make a repartition join between contracts and blocks dataset, where I used the block number as the key to connect them both. This is shown in my code in gasTime.py. Here when running the code the contracts dataset should be the first input followed by the blocks dataset. I read both datasets similar to how I did in my Part B Job 2 code. Where from the blocks dataset I yield the block number as my join key with a compound value of a unique file identifier 2, the gas used for the block, the month the gas was used for that block. From the contracts dataset all I yield is the block number as the key and a value with the file identifier 1 and None.

In my reducer I make a list of contract block numbers where I append all the contract block numbers from the contracts key, value yield. After I will check whether the block number in blocks is in the list of contract block numbers. If yes, I yield the block number followed by a compound value of the gas used and the month, else it will filter out the block and skip to the next block number.



```
"1000690"       [226911, "2016-02"]
"1001581"       [855622, "2016-02"]
"1002322"       [247911, "2016-02"]
"1002544"       [1359506, "2016-02"]
"1002988"       [959694, "2016-02"]
"1003060"       [909440, "2016-02"]
"1003099"       [668609, "2016-02"]
"1003294"       [191309, "2016-02"]
"1003315"       [658284, "2016-02"]
"1003366"       [598682, "2016-02"]
"1003447"       [689673, "2016-02"]
"1003465"       [538003, "2016-02"]
"1003516"       [1338506, "2016-02"]
"1003651"       [696067, "2016-02"]
```

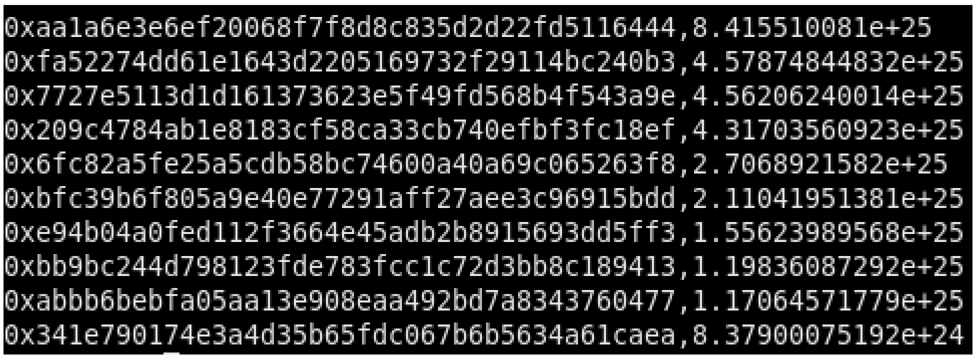
Here is an example of the joined aggregated results

I then needed to reduce the months and get the total amount of gas used for the each month, this is shown in my code gasUsed.py. Here I read my aggregated dataset yielding the month as the key and gas used as the value. In my reduce I reduce the months and sum up the total gas used for that month. I then yield the month as the key and the total gas used for that month as the value. Hence, leading to the values plotted in my graph above.

In regards to the correlation to part B, from the average price of gas prices per month you can see how gas prices have decreased over the months. This leads me to assume that more people will be making more transactions since gas prices are a lot cheaper. Hence the gas usage in the blocks in contracts is higher since, the more transactions that are carried out the more gas needs to be used. And since there are more transactions there is more Wei being transferred, therefore the transaction values are high in part B for the contract blocks.

3. **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)**

In Spark I am reading in inputs from the datasets data/Ethereum/transactions and data/Ethereum/contracts. In reference to my program sparkPartB.py, the is_good_line function checks the dataset being read is the transactions dataset. After this function the transaction dataset is split by fields, the address and the value transferred in Wei is mapped. This is then reduced adding up all the values together with the address as the key. I store this in an RDD and persist, so I can then next read the contracts dataset in the is_good_line2 function. This dataset is split into fields where I then map using the address as the key. I make a join with this mapped address and the RDD results I stored earlier and returns the results with all common keys. Finally I take these results and order them and loop them 10 times to gain the top 10 just like in Part B.



```
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444,8.415510081e+25
0xfa52274dd61e1643d2205169732f29114bc240b3,4.57874844832e+25
0x7727e5113d1d161373623e5f49fd568b4f543a9e,4.56206240014e+25
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef,4.31703560923e+25
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8,2.7068921582e+25
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd,2.11041951381e+25
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3,1.55623989568e+25
0xbb9bc244d798123fde783fcc1c72d3bb8c189413,1.19836087292e+25
0xabbb6bebfa05aa13e908eaa492bd7a8343760477,1.17064571779e+25
0x341e790174e3a4d35b65fdc067b6b5634a61caea,8.37900075192e+24
```

MapReduce times I got: 31mins 59s, 32mins 08s, 35mins 42s, 35mins 39s

MapReduce Average time: 33mins 64s

Spark times I got: 4 mins 26s, 3mins 48s, 3mins 17s, 3mins 14s

Spark Average time: 3mins 51s

For the question "How does it run in comparison?", from the average times I recorded you can see that Spark significantly ran faster than the MapReduce programs (Job1, Job2, Job3). This is because Spark uses in memory-processing(RAM) which gives faster data access than on-disk storage devices, as well as the use of iterative processing (referred to in week 6 lecture). This is good for iterative access and when multiple jobs need the same data. You're also able to save RDDs in the memory meaning you can reuse this later in your code, making the process faster.

So, in Spark for a single file of code I was able to successfully aggregate results from transactions, create a repartition join with transactions and contracts, and aggregate the top 10 results all in one file. Unlike MapReduce, I had to do this in 3 separate files, as 3 separate jobs thus taking longer. I could have done this with MRStep, however this is an extra functionality and it would have been too big to run on the HDFS due to limits with disk quota. Hence making Spark faster.

With the limits of disk quota, the use of multiple files/MapReduce jobs, and the time it takes the overall verdict is that Spark would be better for Part B's task since it was able to rebuke all these issues I came across with MapReduce.