Yulong Cui [190198653]

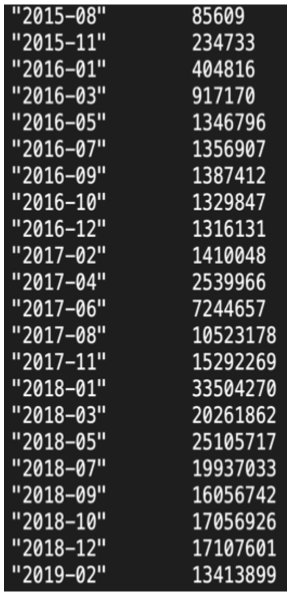
Big Data Coursework: Ethereum Analysis (40%)

Part A – Time Analysis (20%)

1) Number of transactions occurring every month between the start and end of the dataset.

For this section, the map-reduce job is performed on transactions.csv file.

First, in the mapper function, the transaction.csv file is fed into the function line by line. I am splitting each line by comma (‘,’) as the transaction file is a ‘comma separated values’ file. Then I check if the length of the fields after split is 7 as the length of each line is 7. If true, then I am getting the timestamp values from the 6th column, converting it into a readable format ‘Year-Month’ using the time.strftime() function, storing it into ‘date’ variable. Then I am yielding the ‘date’ variable as key with value ‘1’ as a counter, as each time a date appears, it means there is one transaction done.

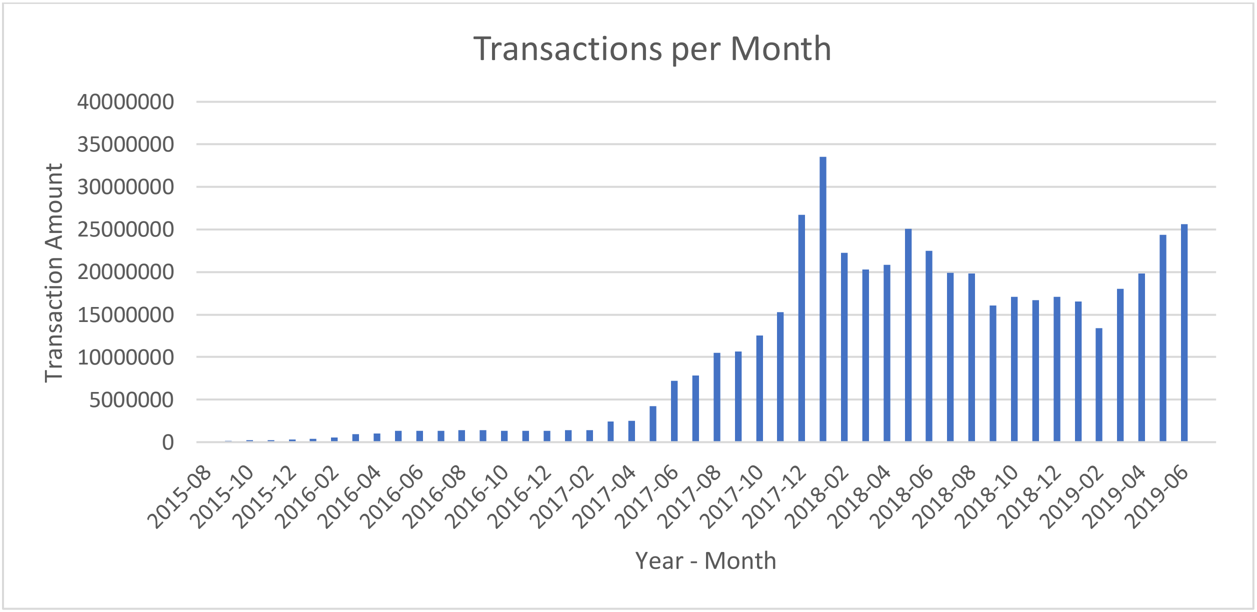


```
"2015-08"       85609
"2015-11"       234733
"2016-01"       404816
"2016-03"       917170
"2016-05"       1346796
"2016-07"       1356907
"2016-09"       1387412
"2016-10"       1329847
"2016-12"       1316131
"2017-02"       1410048
"2017-04"       2539966
"2017-06"       7244657
"2017-08"       10523178
"2017-11"       15292269
"2018-01"       33504270
"2018-03"       20261862
"2018-05"       25105717
"2018-07"       19937033
"2018-09"       16056742
"2018-10"       17056926
"2018-12"       17107601
"2019-02"       13413899
```

Then, in the reducer function, it is receiving key ‘time’ and value ‘value’. I am creating a variable ‘tot’ as a total counter. Then for each ‘time’ variable (format: Year-Month), I am adding ‘value’ variable to the ‘tot’ variable which is essentially counting how many transactions there are in each month. Finally, I am yielding ‘time’ and ‘tot’ to output Year-Month and its corresponding total counts of transactions.

I get output in the form of “Year-Month” and the number of transactions with a tab space separating them (see screenshot of text output).

I used Microsoft Excel to plot the bar graph, with sorted date on the x-axis and total counts of transactions on the y-axis (see screenshot of bar plot below).



2) Average value of transactions in each month between the start and end of dataset.

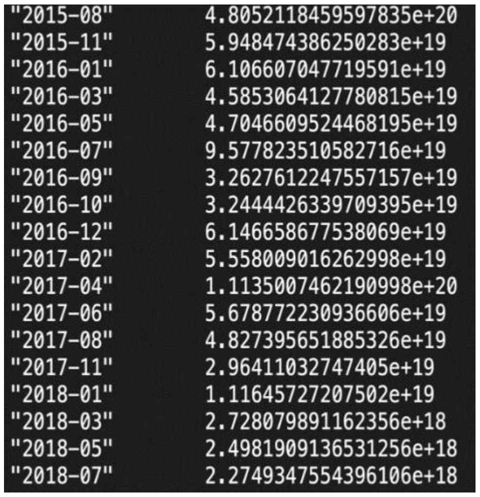
For this section, the map-reduce job is performed on transactions.csv file. For this map-reduce job I have a combiner function that helps reduce the workload for the reducer function.

This question is very similar to A1, instead of finding the total amount of transactions, we are trying to find the actual average values (in Wei) for transactions.

Therefore, in the mapper function, I am doing the exact the same steps to get date. Instead of only yielding a '1' as a counter, I am getting the actual value (transferred in Wei) by using fields[3] and storing it as variable 'amount'.

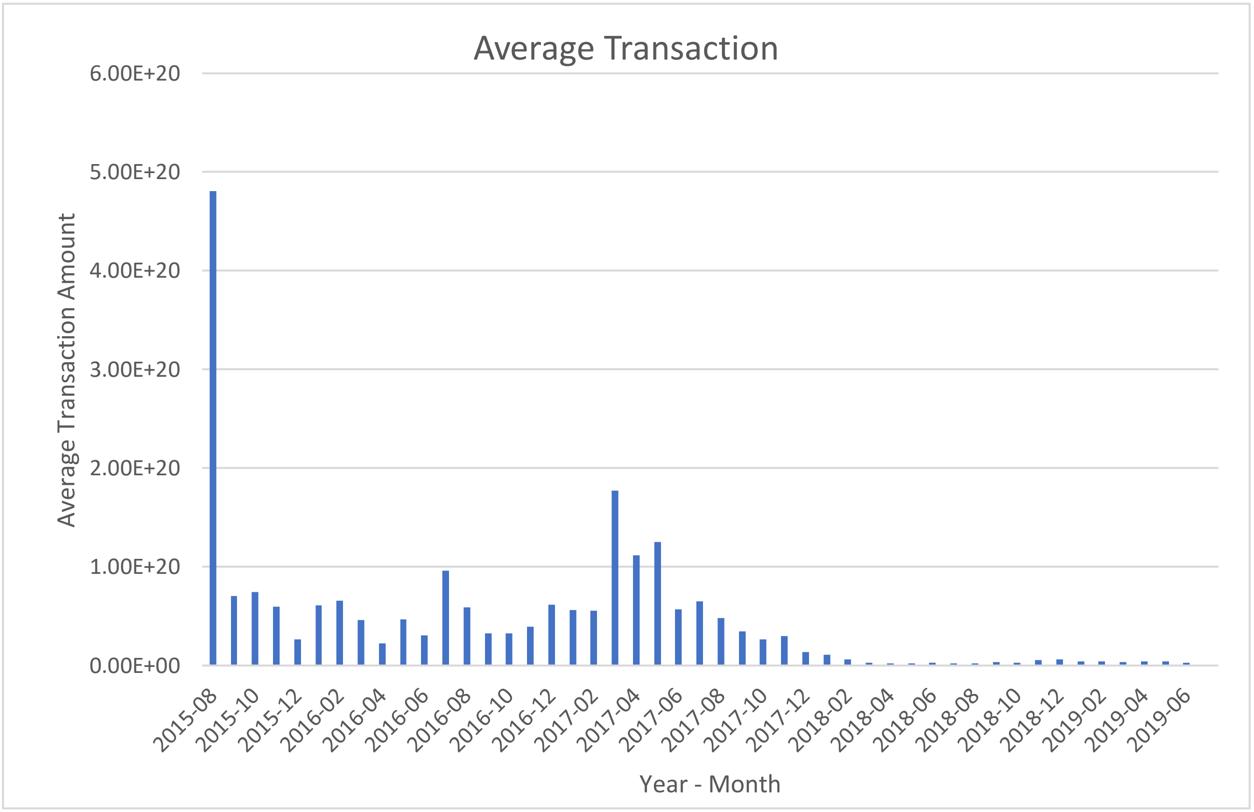
Then, in the combiner function, it is receiving key 'time' and values 'values'. I am creating a counter variable and a total variable. Then for line fed in, I am summing the total value and adding '1' each time. I am yielding 'time' as key and a tuple of its corresponding counter and total values.

Finally, in the reducer, I am summing up the counter variable and total value for every line. Then an 'avg' variable is used to store the average (amount/counter). Finally, I am yielding the average value with time as key. I get output as "Year-month" and average value (See screenshot of text output).



```
"2015-08"       4.8052118459597835e+20
"2015-11"       5.948474386250283e+19
"2016-01"       6.106607047719591e+19
"2016-03"       4.5853064127780815e+19
"2016-05"       4.7046609524468195e+19
"2016-07"       9.577823510582716e+19
"2016-09"       3.2627612247557157e+19
"2016-10"       3.2444426339709395e+19
"2016-12"       6.146658677538069e+19
"2017-02"       5.558009016262998e+19
"2017-04"       1.1135007462190998e+20
"2017-06"       5.678772230936606e+19
"2017-08"       4.827395651885326e+19
"2017-11"       2.96411032747405e+19
"2018-01"       1.11645727207502e+19
"2018-03"       2.728079891162356e+18
"2018-05"       2.4981909136531256e+18
"2018-07"       2.2749347554396106e+18
```

I also used Microsoft Excel to plot this graph with sorted dates on the x-axis and average transaction amount on the y-axis.



Yulong Cui [190198653]

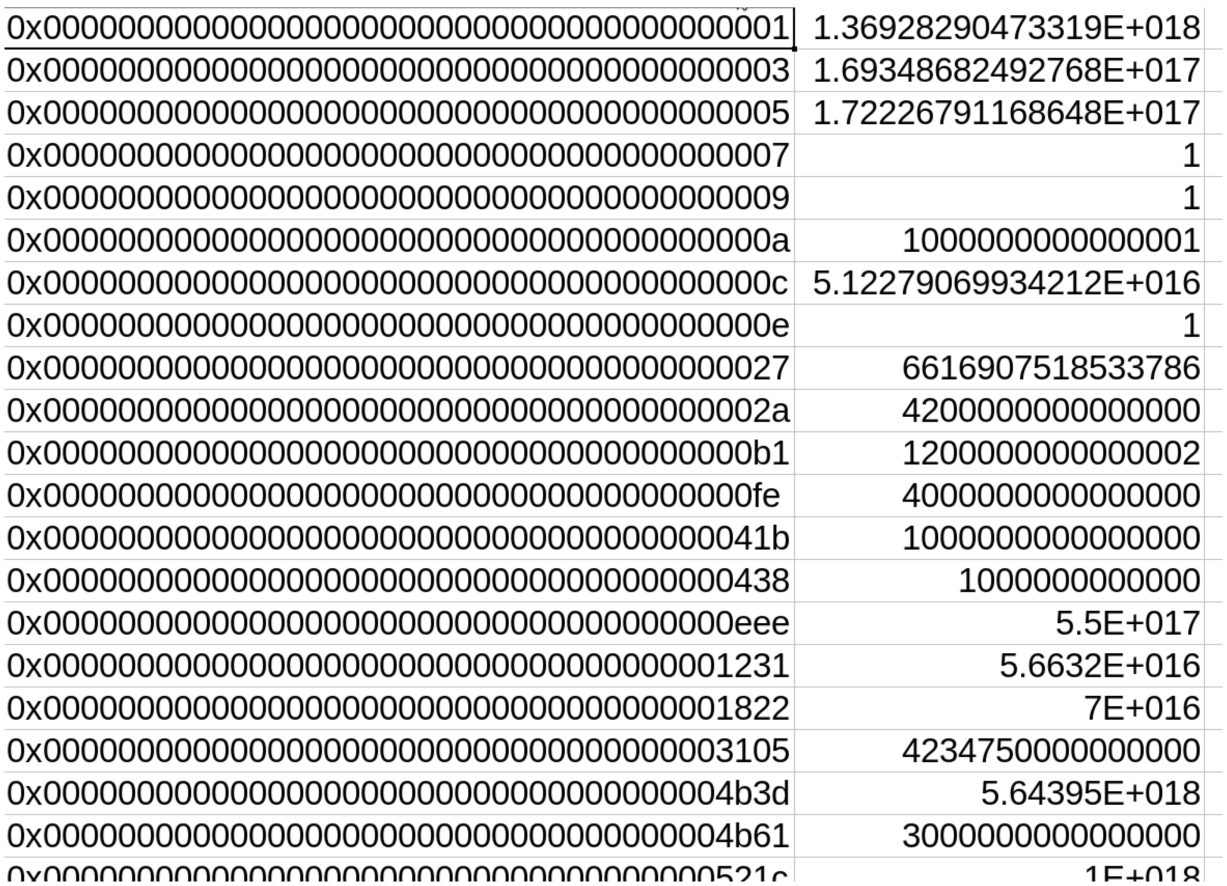
Part B – Top ten most popular services (20%)

1) Initial Aggregation

For this section, we want to aggregate the total value of transactions with its destination address. The Map-Reduce job is performed on the transactions.csv file.

In the mapper function, like in part A, I am splitting the line and getting values by using its index. I am getting the destination address in fields[2] and the transaction value in fields[3]. However, as the transcations.csv file is very big, some of the addresses can be 'null' and some of the values are 0 (as seen in the example table). Therefore, in order to reduce the mapper output size, I am only yielding the address and its corresponding amount if the address is not 'null', and the transactions value is not 0.

In the reducer function, I am summing up the total transaction value and yielding it using its corresponding address as key. The output file for this part is very big (3.4GB in my case, see in screenshot).



0x0000000000000000000000000000000000000001	1.36928290473319E+018
0x0000000000000000000000000000000000000003	1.69348682492768E+017
0x0000000000000000000000000000000000000005	1.72226791168648E+017
0x0000000000000000000000000000000000000007	1
0x0000000000000000000000000000000000000009	1
0x000000000000000000000000000000000000000a	1000000000000001
0x000000000000000000000000000000000000000c	5.12279069934212E+016
0x000000000000000000000000000000000000000e	1
0x0000000000000000000000000000000000000027	6616907518533786
0x000000000000000000000000000000000000002a	4200000000000000
0x00000000000000000000000000000000000000b1	1200000000000002
0x00000000000000000000000000000000000000fe	4000000000000000
0x000000000000000000000000000000000000041b	1000000000000000
0x0000000000000000000000000000000000000438	1000000000000
0x0000000000000000000000000000000000000eee	5.5E+017
0x0000000000000000000000000000000000001231	5.6632E+016
0x0000000000000000000000000000000000001822	7E+016
0x0000000000000000000000000000000000003105	4234750000000000
0x0000000000000000000000000000000000004b3d	5.64395E+018
0x0000000000000000000000000000000000004b61	3000000000000000
0x000000000000000000000000000000000000521c	1E+018

2) Joining transactions/contracts and filtering

For this section, we want to perform a join between two files with address as key and total transaction value as value. The Map-Reduce job is performed on the output file of the previous job (B1) and the contracts.csv file on the Hadoop cluster. Therefore, I had to use -getmerge command line function to download the B1 output file from the Hadoop cluster as a tab-separated-values (.tsv) file onto my local machine and re-upload it onto the Hadoop cluster again for this Map-Reduce job to run. This aim of this section is to filter out the user addresses and only keep the smart contracts.

In the mapper function, I am splitting the lines using comma.

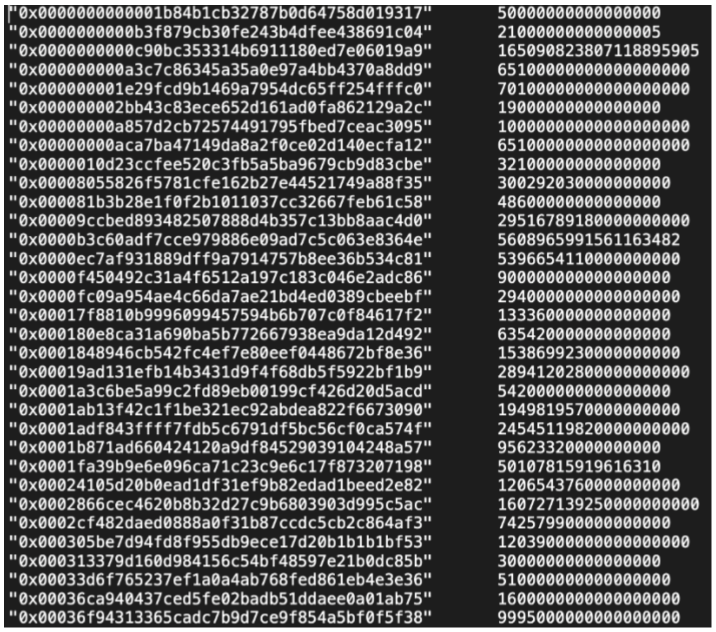
First. I am checking if the length is 1, it means the mapper is receiving a line from my output file from B1. As the length of fields would be 1 if a tab-separated-values file is separated using comma. Then I am splitting the file using tab '\t' and checking if the length is 2, if that is the case then, I am getting the address from fields[0] (with necessary formatting) storing as variable join_key, total transactions value from fields[1] storing as variable join_value. Finally, when I am yielding the results, I am adding a string identifier 'T' to note that this line is from the output file from B1. Yielding a tuple of 'T' and join_value as values with join_key as the key.

Secondly, if the length is 5 when splitting by comma, it means the mapper is receiving a line from the contract.csv file. I am getting the smart contracts address from fields[0] storing as variable join_key. Finally, similar to first part of the mapper function, I am also adding an identifier 'C' to note that this line is from the contract.csv file. Yielding 'C' as values with join_key as key.

In the reducer function, I am creating a tAmount variable to store the total transaction value and a Boolean variable 'exist' to check if the address is a smart contract or just a user address.

For each line the reducer receives if the identifier is 'C', I am setting the Boolean variable 'exist' to true, else (if the the identifier is 'T') I am adding the join_value onto the total transaction value.

Finally yielding only if the address is a smart contract and its value is not 0. (See output screenshot)

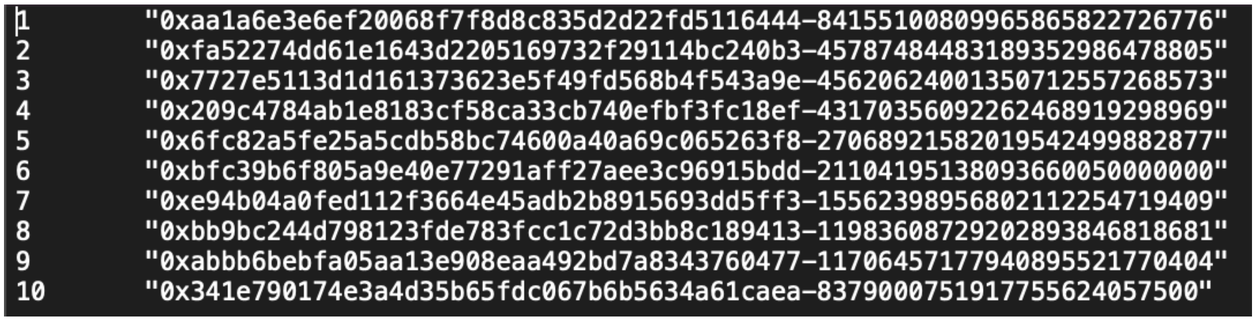

```
"0x0000000000001b84b1cb32787b0d64758d019317"      50000000000000000
"0x0000000000b3f879cb30fe243b4dfee438691c04"      21000000000000005
"0x0000000000c90bc353314b6911180ed7e06019a9"      165090823807118895905
"0x000000000a3c7c86345a35a0e97a4bb4370a8dd9"      651000000000000000000
"0x000000001e29fcd9b1469a7954dc65ff254fffc0"      701000000000000000000
"0x000000002bb43c83ece652d161ad0fa862129a2c"      19000000000000000
"0x00000000a857d2cb72574491795fbed7ceac3095"      100000000000000000000
"0x00000000aca7ba47149da8a2f0ce02d140ecfa12"      651000000000000000000
"0x0000010d23ccfee520c3fb5a5ba9679cb9d83cbe"      32100000000000000
"0x00008055826f5781cfe162b27e44521749a88f35"      300292030000000000
"0x000081b3b28e1f0f2b1011037cc32667feb61c58"      48600000000000000
"0x00009ccbed893482507888d4b357c13bb8aac4d0"      295167891800000000000
"0x0000b3c60adf7cce979886e09ad7c5c063e8364e"      5608965991561163482
"0x0000ec7af931889dff9a7914757b8ee36b534c81"      5396654110000000000
"0x0000f450492c31a4f6512a197c183c046e2adc86"      900000000000000000
"0x0000fc09a954ae4c66da7ae21bd4ed0389cbeebf"      2940000000000000000
"0x00017f8810b9996099457594b6b707c0f84617f2"      133360000000000000
"0x000180e8ca31a690ba5b772667938ea9da12d492"      635420000000000000
"0x0001848946cb542fc4ef7e80eef0448672bf8e36"      1538699230000000000
"0x00019ad131efb14b3431d9f4f68db5f5922bf1b9"      28941202800000000000
"0x0001a3c6be5a99c2fd89eb00199cf426d20d5acd"      542000000000000000
"0x0001ab13f42c1f1be321ec92abdea822f6673090"      1949819570000000000
"0x0001adf843ffff7fdb5c6791df5bc56cf0ca574f"      24545119820000000000
"0x0001b871ad660424120a9df84529039104248a57"      95623320000000000
"0x0001fa39b9e6e096ca71c23c9e6c17f873207198"      50107815919616310
"0x00024105d20b0ead1df31ef9b82edad1beed2e82"      1206543760000000000
"0x0002866cec4620b8b32d27c9b6803903d995c5ac"      160727139250000000000
"0x0002cf482daed0888a0f31b87ccdc5cb2c864af3"      742579900000000000
"0x000305be7d94fd8f955db9ece17d20b1b1b1bf53"      12039000000000000000
"0x000313379d160d984156c54bf48597e21b0dc85b"      30000000000000000
"0x00033d6f765237ef1a0a4ab768fed861eb4e3e36"      510000000000000000
"0x00036ca940437ced5fe02badb51ddaee0a01ab75"      1600000000000000000
"0x00036f94313365cadc7b9d7ce9f854a5bf0f5f38"      9995000000000000000
```

3) Top ten

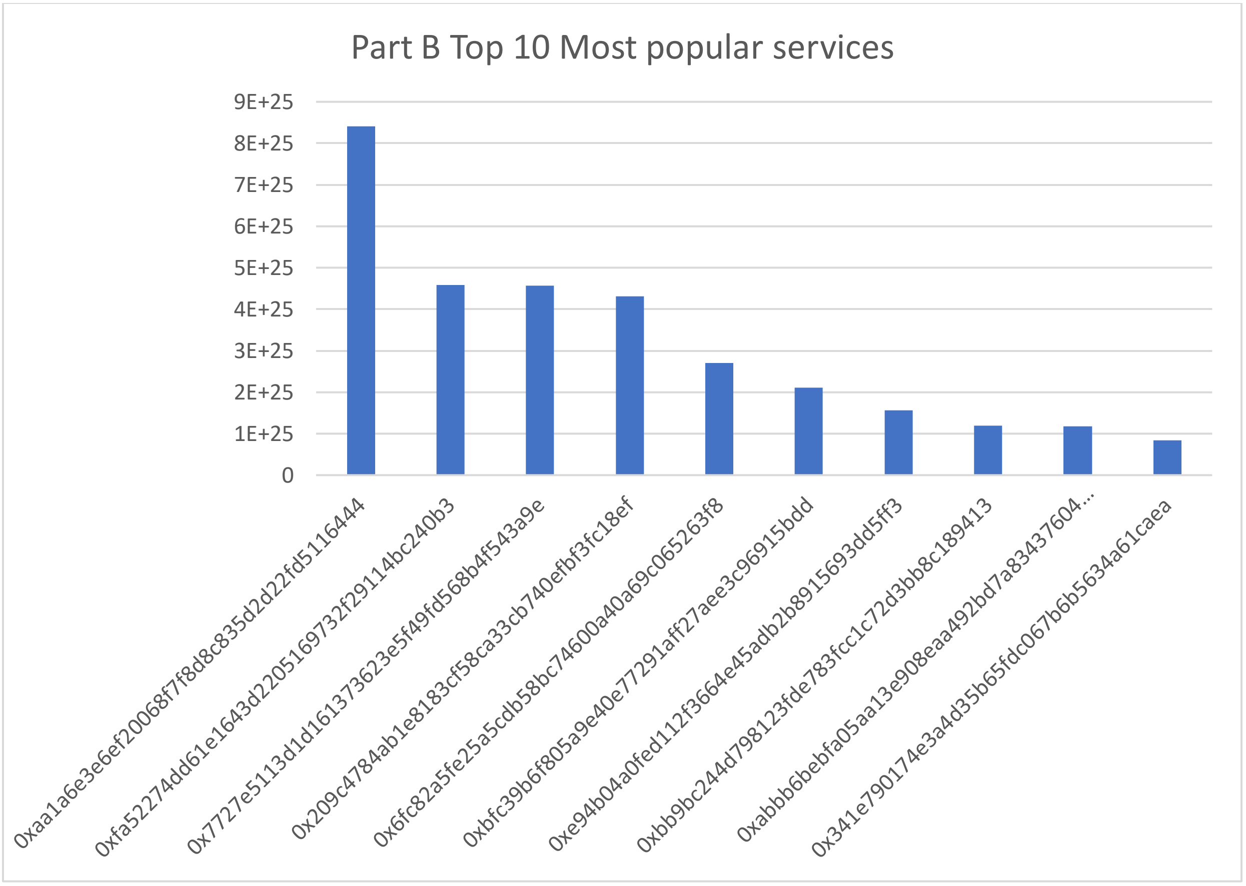
For this section, we want to perform a sort by join_value (total transaction value) and only yield the top 10 results. This Map-Reduce job was done locally as my python file and the output result from B2 were both stored locally.

In the mapper function, I am getting the address from fields[0] and total transaction value from fields[1]. Yielding it to the reducer with total transaction and address in a tuple as value with no key (None).

In the reducer function, I created a counter variable to record the index of the top 10 most popular services. Then I used a sort function that sorts by the transaction values from the highest to the lowest and gets the top ten values using array slicing. Finally, yielding a string formatted tuple of address and total transaction with '-' splitting them, the index counter as key. (See output screenshot)



```
1        "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444-84155100809965865822726776"
2        "0xfa52274dd61e1643d2205169732f29114bc240b3-45787484483189352986478805"
3        "0x7727e5113d1d161373623e5f49fd568b4f543a9e-45620624001350712557268573"
4        "0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef-43170356092262468919298969"
5        "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8-27068921582019542499882877"
6        "0xbfc39b6f805a9e40e77291aff27aee3c96915bdd-21104195138093660050000000"
7        "0xe94b04a0fed112f3664e45adb2b8915693dd5ff3-15562398956802112254719409"
8        "0xbb9bc244d798123fde783fcc1c72d3bb8c189413-11983608729202893846818681"
9        "0xabbb6bebfa05aa13e908eaa492bd7a8343760477-11706457177940895521770404"
10       "0x341e790174e3a4d35b65fdc067b6b5634a61caea-8379000751917755624057500"
```



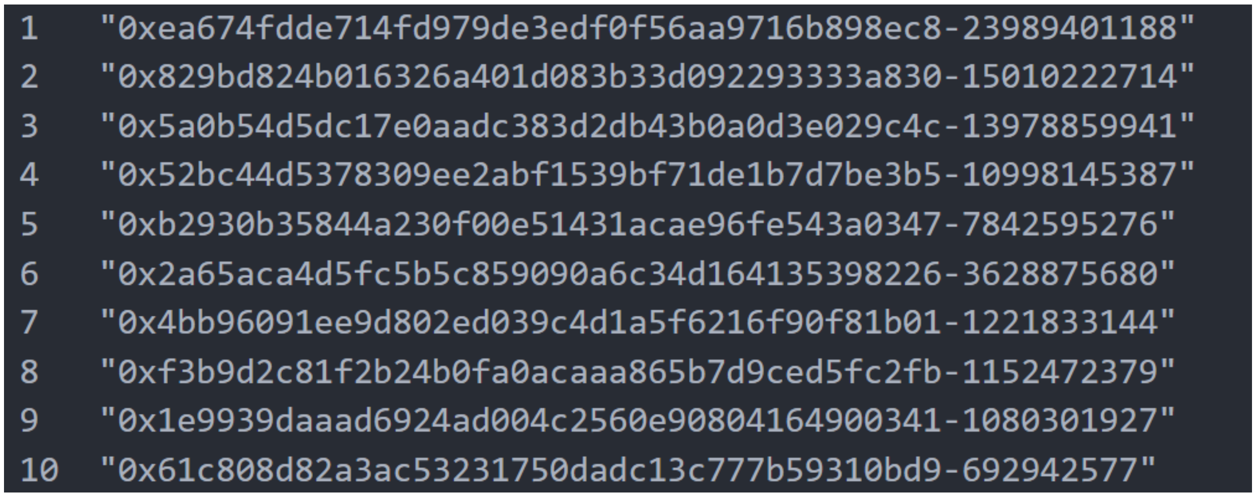
Yulong Cui [190198653]

Part C - Top ten most active miners (10%)

For this section, the Map-Reduce job is performed on the block.csv file on the Hadoop cluster. We want to find the top ten most active miners, which means, we want to find the miners with the highest ‘size’ value in the block.csv file. For this job, I am using MRStep with two mappers and two reducer functions.

In the first Map-Reduce job, I am trying to find all the miners and their total block size. In the mapper function, I am splitting the input lines by comma and checking if the length is 9. If true, I am getting the miner address from index 2 storing as variable ‘address’ and the size of the block from index 4 storing as variable ‘amount’. Yielding amount as value with key address. Then in the reducer function, I created a variable ‘tot’ to store the total size of the blocks for each miner address. Yielding tot as value and address as key.

In the second Map-Reduce job, I am trying to find the top ten active miners. This is almost identical to the third question in part B. In the mapper, I am just yielding a tuple of tot and address as value and no key (None). Finally, in the reducer, I am creating an index counting variable ‘count’, set it to 0. Then, I used a sort function that sorts by the total block size from the highest to the lowest and gets the top ten values using array slicing. Finally, yielding a string formatted tuple of address and total transaction with ‘-‘ splitting them, the index counter as key. (See output screenshot)



```
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"
```

Yulong Cui [190198653]

Part D – Data Exploration (50%)

Miscellaneous Analysis

1) Fork the chain

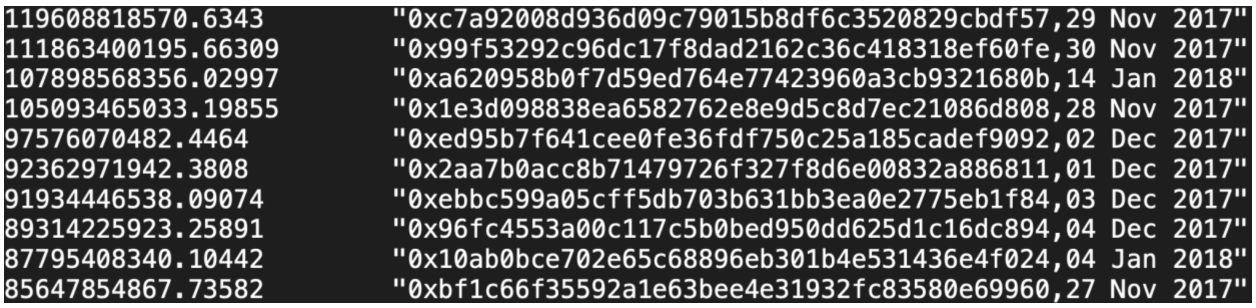
For this section, we want to find what effects does forking of Ethereum have on the price and general usage. I did some research about forking Ethereum and chose the Byzantium fork[i] on the 16th of October 2017 as the example, I did my analysis over a three-month period (16th Oct 2017 to 16th Jan 2018) to see which user has profited the most. I have separated the Map-Reduce job into two different files, first one will get all the valid addresses, transactions values and corresponding date. The second job is getting the top ten using the output of the first job.

The first Map-Reduce job is performed on transactions.csv file. In the mapper function, I am splitting the lines fed into the mapper by comma, getting trader address from index 2, transaction value from index 3 and timestamp from index 6. Then, I have a Boolean variable 'isBetween' to decide whether the transaction done is during the three months period. I am also checking if the address is not 'null' and the transaction value is not 0. If all three conditions are satisfied then I am yielding a tuple of address, transaction value as values with formatted date as key.

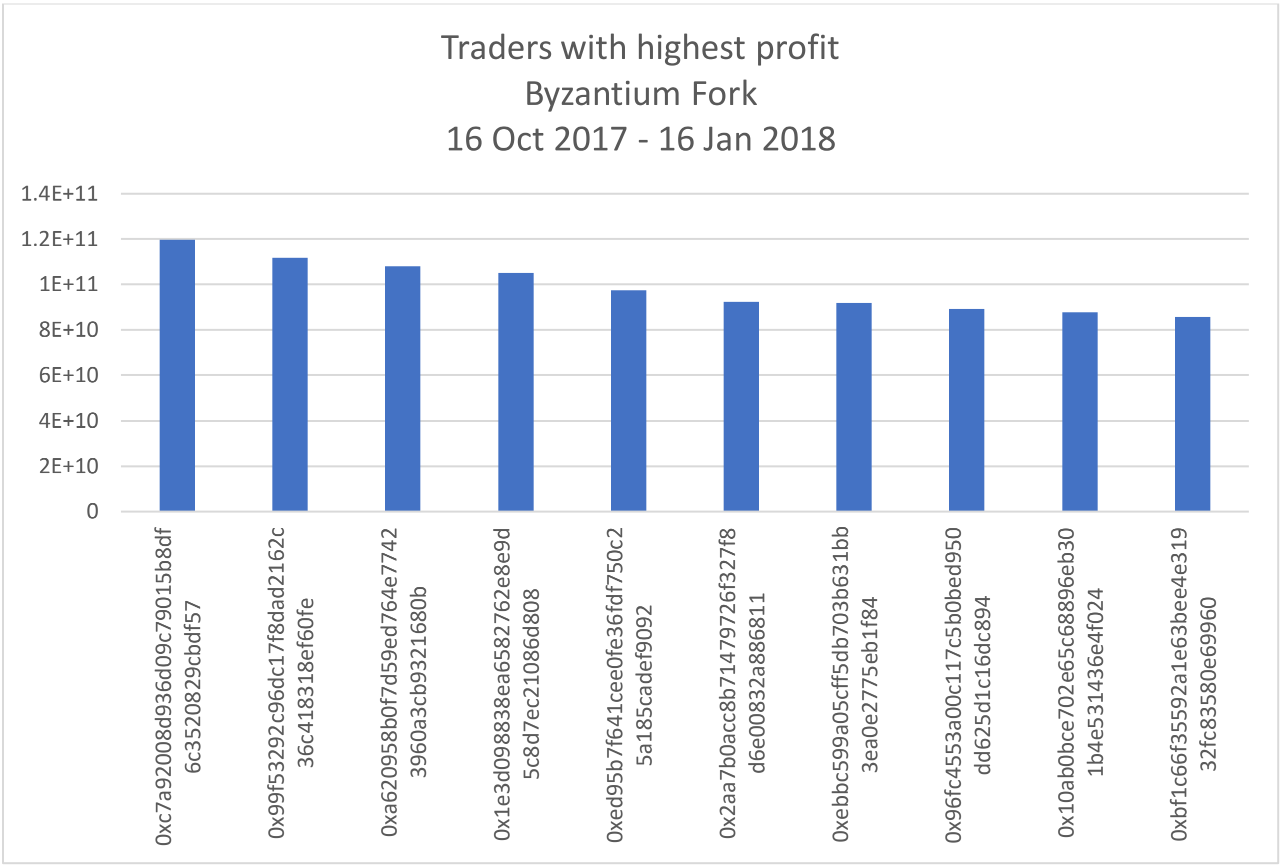
In the reducer function, I have a total transaction variable 'tot' and for each line fed in, I am summing up the transaction value. Then, I have another total transaction variable 'totInUSD' which converts the transaction value from unit 'Wei' to US dollars. Finally, I am yielding a tuple of transaction value in USD, transaction date as values and trader address as key.

The second Map-Reduce job performed on the output file of the first Map-Reduce job. Again, it is very similar to other top-ten jobs that I have done in the coursework. I am splitting the fields using tabs, getting the address from index 0, an array of values (transaction value, date) from index 1. Then I am splitting the array using comma, getting the transaction value from index 0 and date from index 1. I am doing this to properly format the transaction value before sorting it as the sort function cannot sort string inputs. Yielding address and a tuple of transaction value and date with no key (None).

In the reducer, I am sorting using the transaction value in dollars, getting the top ten values using slicing. Yielding it with date as values and trader address as key (See screenshot below)



```
119608818570.6343       "0xc7a92008d936d09c79015b8df6c3520829cbdf57,29 Nov 2017"
111863400195.66309      "0x99f53292c96dc17f8dad2162c36c418318ef60fe,30 Nov 2017"
107898568356.02997      "0xa620958b0f7d59ed764e77423960a3cb9321680b,14 Jan 2018"
105093465033.19855      "0x1e3d098838ea6582762e8e9d5c8d7ec21086d808,28 Nov 2017"
97576070482.4464        "0xed95b7f641cee0fe36fdf750c25a185cadef9092,02 Dec 2017"
92362971942.3808        "0x2aa7b0acc8b71479726f327f8d6e00832a886811,01 Dec 2017"
91934446538.09074       "0xebbc599a05cff5db703b631bb3ea0e2775eb1f84,03 Dec 2017"
89314225923.25891       "0x96fc4553a00c117c5b0bed950dd625d1c16dc894,04 Dec 2017"
87795408340.10442       "0x10ab0bce702e65c68896eb301b4e531436e4f024,04 Jan 2018"
85647854867.73582       "0xbf1c66f35592a1e63bee4e31932fc83580e69960,27 Nov 2017"
```


Traders with highest profit
Byzantium Fork
16 Oct 2017 - 16 Jan 2018
1.4E+11
1.2E+11
1E+11
8E+10
6E+10
4E+10
2E+10
0
0xc7a92008d936d09c79015b8df6c3520829cbdf57
0x99f53292c96dc17f8dad2162c36c418318ef60fe
0xa620958b0f7d59ed764e77423960a3cb9321680b
0x1e3d098838ea6582762e8e9d5c8d7ec21086d808
0xed95b7f641cee0fe36fdf750c25a185cadef9092
0x2aa7b0acc8b71479726f327f8d6e00832a886811
0xebbc599a05cff5db703b631bb3ea0e2775eb1f84
0x96fc4553a00c117c5b0bed950dd625d1c16dc894
0x10ab0bce702e65c68896eb301b4e531436e4f024
0xbf1c66f35592a1e63bee4e31932fc83580e69960

2) Gas Guzzlers

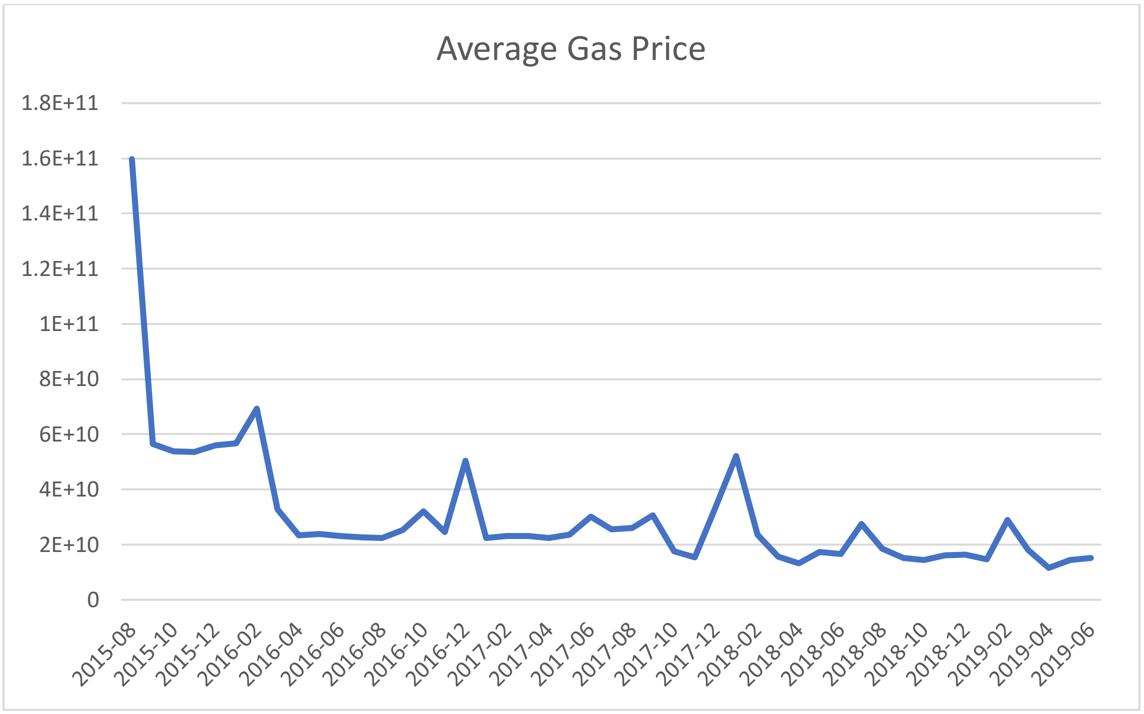
For this section, we want to find out how the average gas price changed over time and whether the contracts have become more complicated. I did both questions in Spark.

I am parsing in all the 'dirty' unfiltered lines from transaction.csv, block.csv and contracts.csv file. Then I have three functions format the input lines according to the variables I need.

a) Average Gas Prices

For this question, I am getting the gas price from index 5 and timestamp from index 6 from the 'clean' filtered transactions lines. Then I am string-formatting the timestamp and adding a counter '1'. Finally, in the reduceByKey function, I am reducing by calculating the average gas price by dividing total gas price on a date by the counter and outputting by sorting it from lowest to highest with date as key. I have plotted the outputs in Excel (See screenshot of plotted line graph below).

As seen in the graph, the average gas price has decreased over time. This means the average transaction fees are decreasing over time which makes sense as the cryptocurrency market is becoming more and more popular, more people started trading cryptocurrencies such as Bitcoin and Ethereum. Therefore, the transaction fees have decreased.

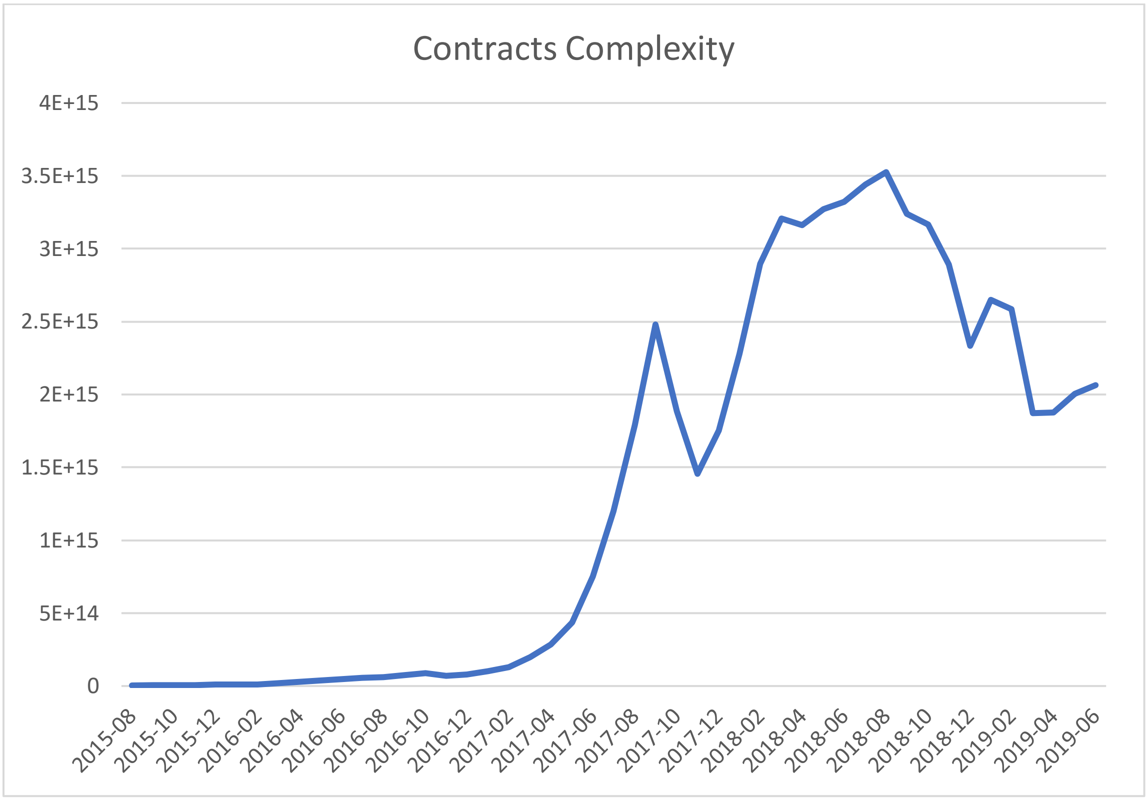


b) Contract Complexity

For this question, we want to find out if the difficulty level of smart contracts has increased or decreased over time.

First, I am getting values from ‘clean’ filtered contracts lines, block number from index 3 and a counter ‘1’ for each block. Then I am getting values from ‘clean’ filtered block lines, block number from index 0, a tuple of values: difficulty from index 3, gas used from index 6, and formatted date (timestamp) from index 7. Then, I am joining the previous two outputs using block number as key to check if the contract from the block.csv file is in the contracts.csv file. Finally, in the reduceByKey function, I am outputting date, average difficulty, and the corresponding block number. Sorted by the average difficulty from lowest to the highest. I have plotted the output with date on the x-axis and average difficulty on the y-axis (see in screenshot).

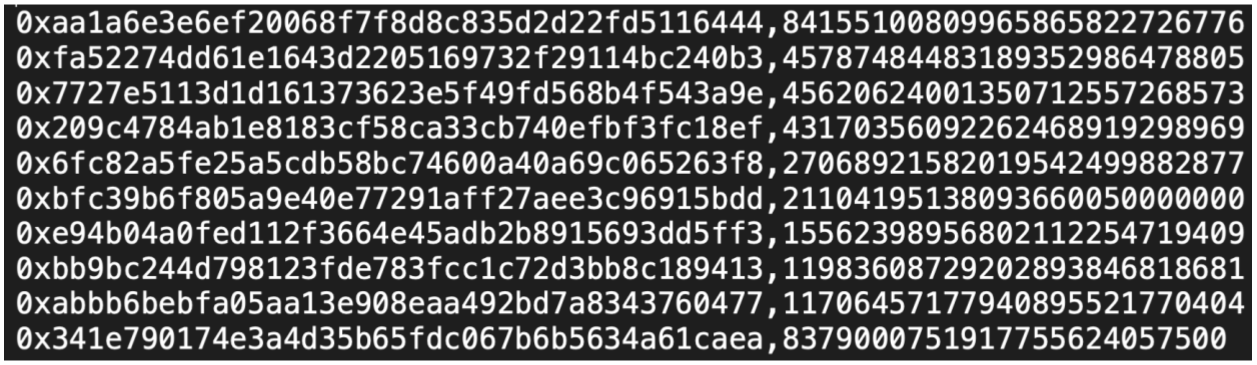
As seen in the line graph, the complexity of contracts has become more and more complex over time. This makes sense as the cryptocurrency technologies are developing over time, allowing Ethereum to do more complicated contracts. There are also certain forks of the Ethereum that can increase the complexity of contracts.



c) Comparative Evaluation

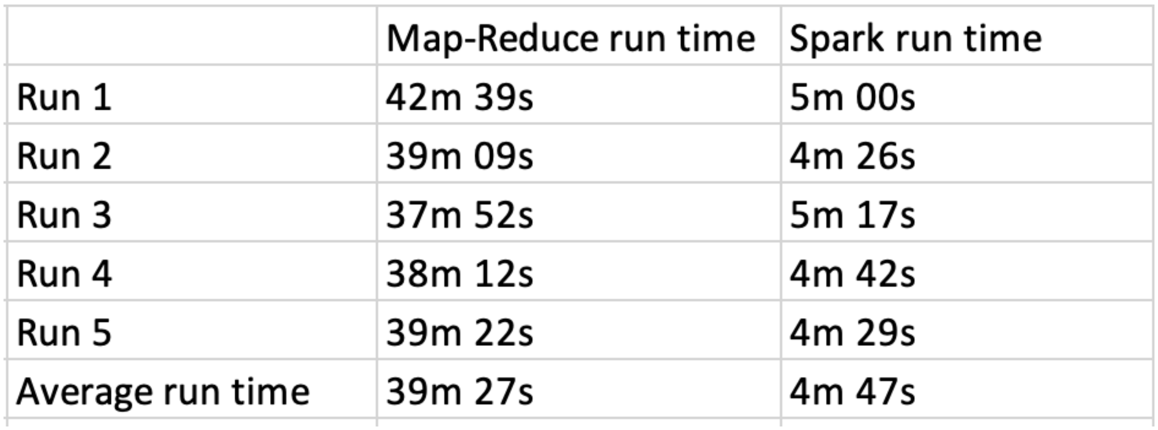
For this section, we are comparing the time it takes to run Part B using Spark and Map-Reduce.

For the Spark version of Part B, I am doing the same filtering processes as the previous (part D Misc B) for transactions.csv and contracts.csv. I am getting values from transaction lines: address(key) from index 2 and transaction value(value) from index 3. Then, summing up the transaction values for each address using the 'add' operator. Then, joining with the address(index 0) from contracs.csv. Finally using takeOrdered function, sorting from highest to lowest using total transaction value, taking top ten with address as key and total transaction value as value(see screenshot of Spark output from terminal).



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

I ran Part B in Map-Reduce and Spark 5 time each and here are the recorded times (see in screenshot of table). On average, Map-Reduce took 39 minutes 27 seconds to run and Spark took 4 minutes 47 seconds to run. This is because I have three separate files to run for the Map-Reduce job whereas for Spark I only have one file to run. Also, as Spark utilizes existing RDDs from previous jobs, it also uses in-memory processing and iterative processing to drastically reduce the run time. Therefore, the overall run time for Spark is much less than Map-Reduce.



	Map-Reduce run time	Spark run time
Run 1	42m 39s	5m 00s
Run 2	39m 09s	4m 26s
Run 3	37m 52s	5m 17s
Run 4	38m 12s	4m 42s
Run 5	39m 22s	4m 29s
Average run time	39m 27s	4m 47s

[i] History and forks of ethereum | ethereum.org [online]. *ethereum.org*. [Viewed 3 December 2021]. Available from: https://ethereum.org/en/history/