Ethereum Analysis

Shivam Tyagi †
School of Electronic Engineering and Computer Science
Queen Mary University of London Mile End
Road, London E1 4NS, UK
s.tyagi@se22.qmul.ac.uk

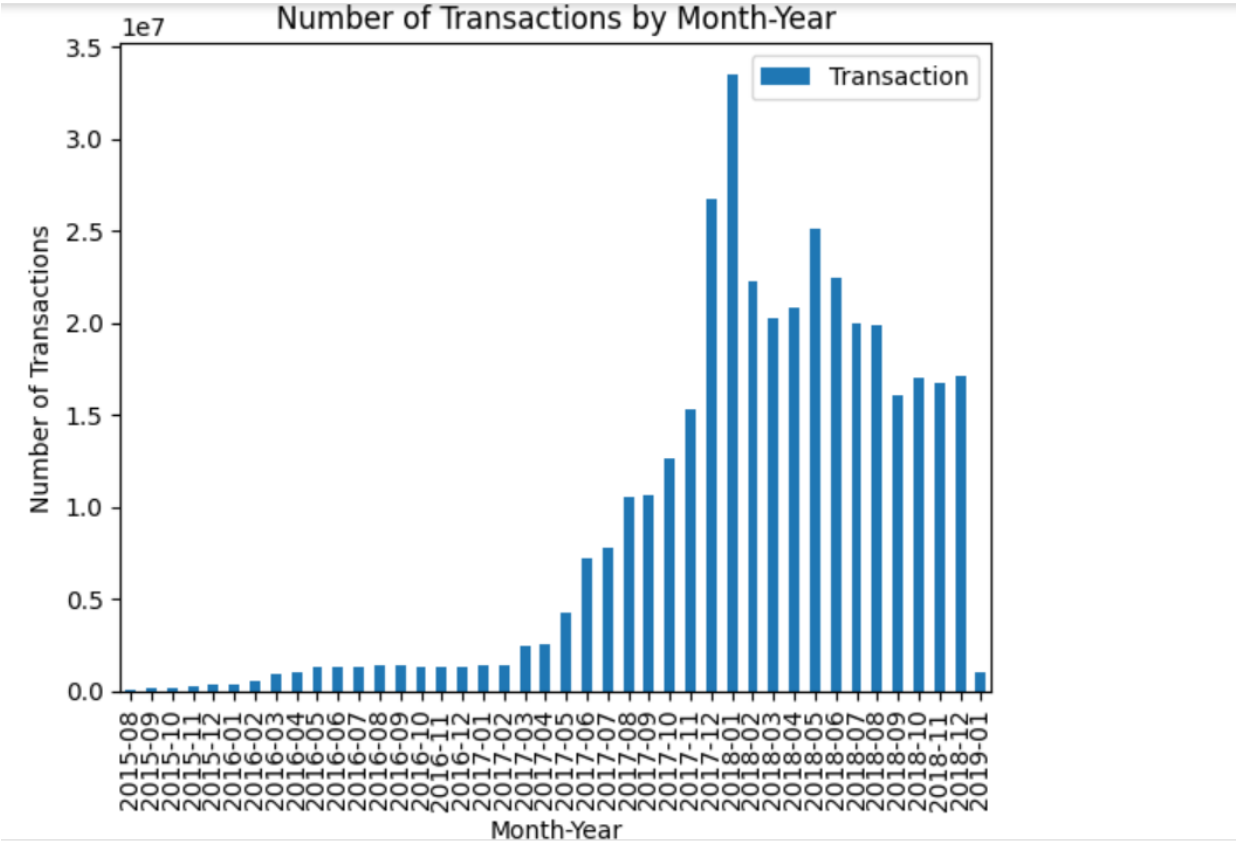
Abstract

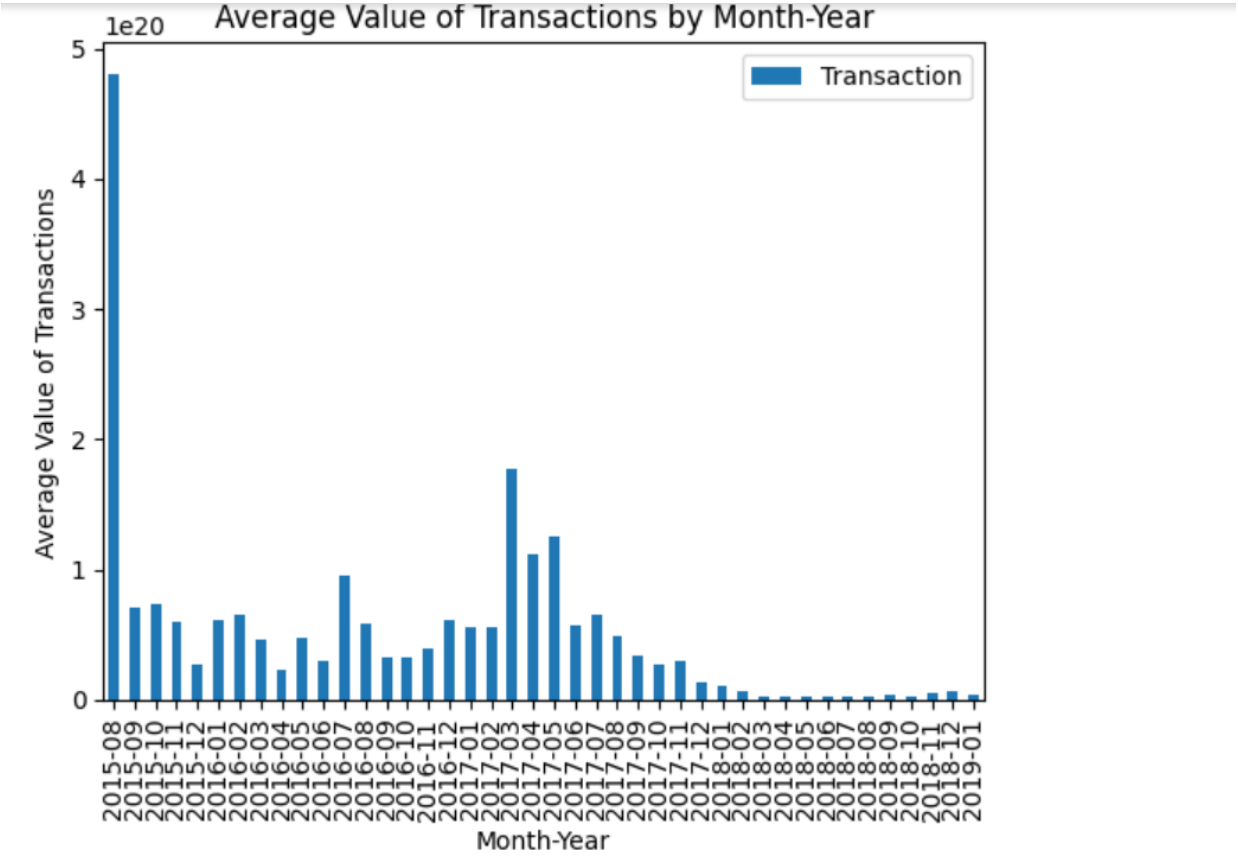
The goal of this project is to analyse Ethereum transactions and smart contracts using Big Data Processing techniques covered in the Big Data Processing course. The analysis covers the full set of transactions that took place on the network from August 2015 to January 2019. The project uses a dataset containing information about blocks, transactions, and contracts on the Ethereum network, as well as a dataset of known scams on the network. The blocks dataset includes information such as block number, hash, nonce, gas limit, and transaction count. The transactions dataset includes information such as transaction hash, sender and receiver addresses, gas price, and value transferred. The contracts dataset includes information about contract addresses, bytecode, and function signature hashes. The scams dataset includes information about reported scams, including their name, hosting URL, associated addresses, and status.

The project requires the creation of several Spark programs to perform various computations on the dataset.

1 Part A

- The first step was to filter out any invalid lines from the transactions.csv file by creating a function called good_line. This function checks if the number of fields in each line is 15, and if the timestamp field is a valid integer. Any line that fails these checks is filtered out.
- The next step was to read the clean transaction lines from the CSV file using Spark's textFile method, and then apply the good_line function to filter out any invalid lines.
- For the first requirement, a map operation was performed on the clean_lines RDD to extract the year-month information from the timestamp field using the strftime function from the time library. The result was a key-value pair RDD where the key was the year-month and the value was 1 to indicate the occurrence of a transaction in that month. Then, a reduceByKey operation was used to sum up the occurrence count for each year-month key. This produced an RDD with year-month keys and transaction count values.
- For the second requirement, another map operation was performed on the clean_lines RDD to extract the year-month information and the value of the transaction. The result was a key-value pair RDD where the key was the year-month and the value was a tuple containing the value of the transaction and a count of 1 to indicate the occurrence of a transaction in that month. Then, a reduceByKey operation was used to sum up the transaction values and the transaction count for each year-month key. This produced an RDD with year-month keys and the average value of the transaction in each month.
- Finally, the results were written to CSV files and saved to an S3 bucket using the put method from the S3 resource object.
- The CSV's were read in python and the bar plot was plotted after sorting the dates and the results were as follows (the ipynb file used to process the same is in the zip folder with the name: **Part A.ipynb**):





2 Part B

To Evaluate Top 10 smart Contracts by total Ether received, the following steps were taken:

- The first step was to filter out any invalid lines from the transactions.csv and contracts.csv files by creating two functions called good_line1 and good_line2. good_line1 function checks if the

number of fields in each line is 15, if the address field is a valid string and if the timestamp field is a valid integer for transaction dataset, good_line2 function checks if the number of fields in each line is 6 and if the address field is a valid string for the contract dataset. Any line that fails these checks is filtered out.

- The next step was to read the clean transaction and contract lines from the CSV file using Spark's textFile method, and then apply the two good_line functions to filter out any invalid lines.
- The transactions dataset is mapped to a tuple containing the transaction address as key and value of transaction as the value, and then reduced by key to sum up the values of all transactions with the same address.
- The clean_lines1 dataset (which contains the cleaned contracts data) is then mapped to contain contract address as key and the string 'contract' as value, the resulting RDD is then joined with the tran_red dataset (which contains the summed transaction values by address) using the address field as the key.
- Finally, the top 10 contracts are selected by taking the 10 largest values in the joined dataset using the takeOrdered function.
- The output is shared as a txt file: **top10.txt** in the zip folder.


```
[["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444", [84155363699941767867374641, "contract"]],
["0x7727e5113d1d161373623e5f49fd568b4f543a9e", [45627128512915344587749920, "contract"]],
["0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef", [42552989136413198919298969, "contract"]],
["0xbfc39b6f805a9e40e77291aff27aee3c96915bdd", [21104195138093660050000000, "contract"]],
["0xe94b04a0fed112f3664e45adb2b8915693dd5ff3", [15543077635263742254719409, "contract"]],
["0xabbb6bebfa05aa13e908eaa492bd7a8343760477", [10719485945628946136524680, "contract"]],
["0x341e790174e3a4d35b65fdc067b6b5634a61caea", [837900075191775624057500, "contract"]],
["0x58ae42a38d6b33a1e31492b60465fa80da595755", [29027091871057365328638 18, "contract"]],
["0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3", [12380861145200420000000000, "contract"]],
["0xe28e72fcf78647adce1f1252f240bbfaebd63bcc", [11724264325158231427145 82, "contract"]]]
```


3 Part C

To evaluate the top 10 miners by the size of the blocks mined, the following steps were taken:

- The first step was to filter out any invalid lines from the blocks.csv file by creating a function called good_line. This function checks if the number of fields in each line is 19, and if the address field is a valid string. Any line that fails these checks is filtered out.
- The next step was to read the clean block lines from the CSV file using Spark's textFile method, and then apply the good_line function to filter out any invalid lines.
- The next step was to map the data to include only the address of the miner and the size of the block. Then, the data was aggregated by summing the sizes of each block for each miner using the reduceByKey function.
- Finally, to obtain the top 10 miners by size, the list of miners was sorted by size and the first 10 elements were taken using the takeOrdered function. This produced a list of the top 10 miners by block size.
- The output is shared as a txt file: **top10-Miners.txt** in the zip folder.

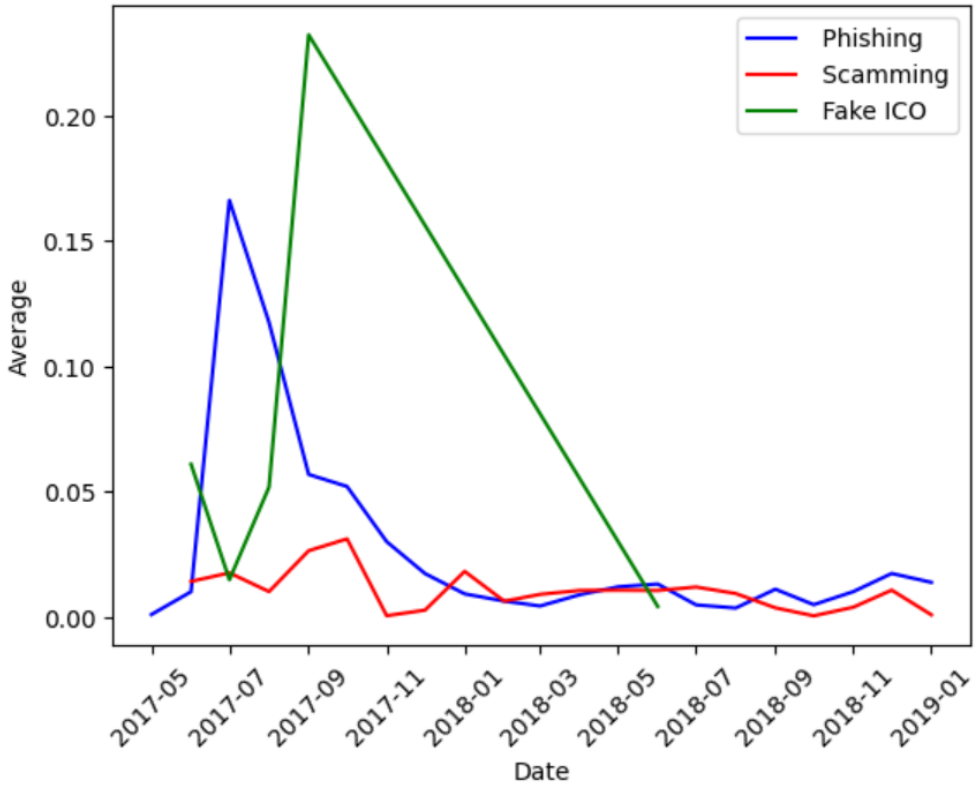

```
[["0xea674fdde714fd979de3edf0f56aa9716b898ec8", 17453393724], ["0x829bd824b016326a401d083b33d092293333a830",
12310472526], ["0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c", 8825710065],
["0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5", 8451574409], ["0xb2930b35844a230f00e51431acae96fe543a0347",
6614130661], ["0x2a65aca4d5fc5b5c859090a6c34d164135398226", 3173096011],
["0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb", 1152847020], ["0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01",
1134151226], ["0x1e9939daaad6924ad004c2560e90804164900341", 1080436358],
["0x61c808d82a3ac53231750dadc13c777b59310bd9", 692942577]]
```


4 Part D – Scam Analysis

- The first step was to filter out any invalid lines from the transactions.csv and scams.csv (note that it is different from the schema in json and was verified by experimenting with the csv that total fields

are 8 and their position was also determined) file by creating two functions called good_line1 and good_line2. good_line1function checks if the number of fields in each line is 15, if the address field is a valid string and if the timestamp field is a valid integer for transaction dataset, good_line2 function checks if the number of fields in each line is 8, if the category field is a valid string, if the ID field is a valid string and if the address field is a valid string for the scam dataset. Any line that fails these checks is filtered out.

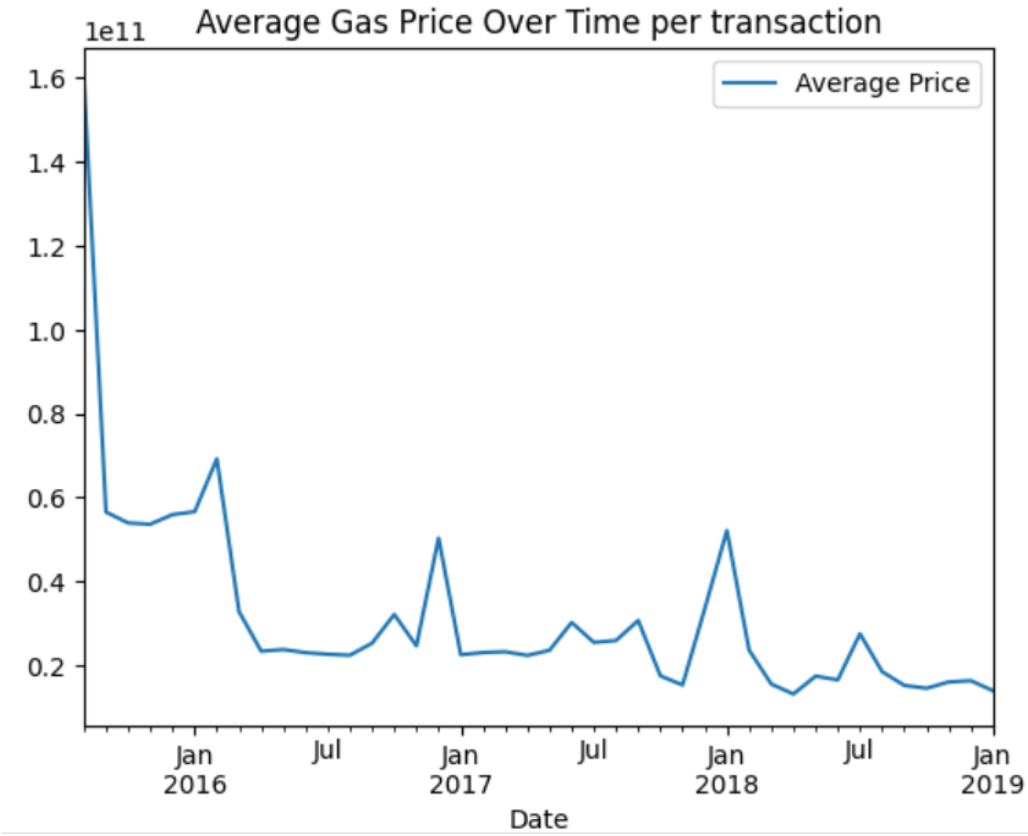
- The next step was to read the clean transaction and scam lines from the CSV file using Spark's textFile method, and then apply the two good_line functions to filter out any invalid lines.
- The code then creates two RDDs, one for transactions and one for scams. The transactions RDD is mapped to extract the address as key and value, timestamp, and count fields as Values. The timestamp is converted to Year-month format using time.strftime() function. The scams RDD is mapped to extract the address as key and type of scam, and ID fields as values.
- The two RDDs are joined on the address field using the join () function. This creates a new RDD that contains the address along with the value, time, and count of the transaction as well as the type and ID of the scam.
- The RDD is then mapped to create two new RDDs, tscomb1 and tscomb2. The tscomb1 RDD uses the type of scam and timestamp fields as keys with the value and count fields as values. The tscomb2 RDD uses the ID field as the key and the value and count fields as values.
- The first RDD is then reduced by key using the reduceByKey() function and then mapped to take the type as key and timestamp, value and count as values and the second RDD is then reduced by key using the reduceByKey() function. The corresponding outputs out1 and out2 are then sent out as txt files: scams.txt and most-lucarative.txt. (Note: The values in the RDD's are divided by 10^20 to make them easier to plot).
- The corresponding plots are as follows:

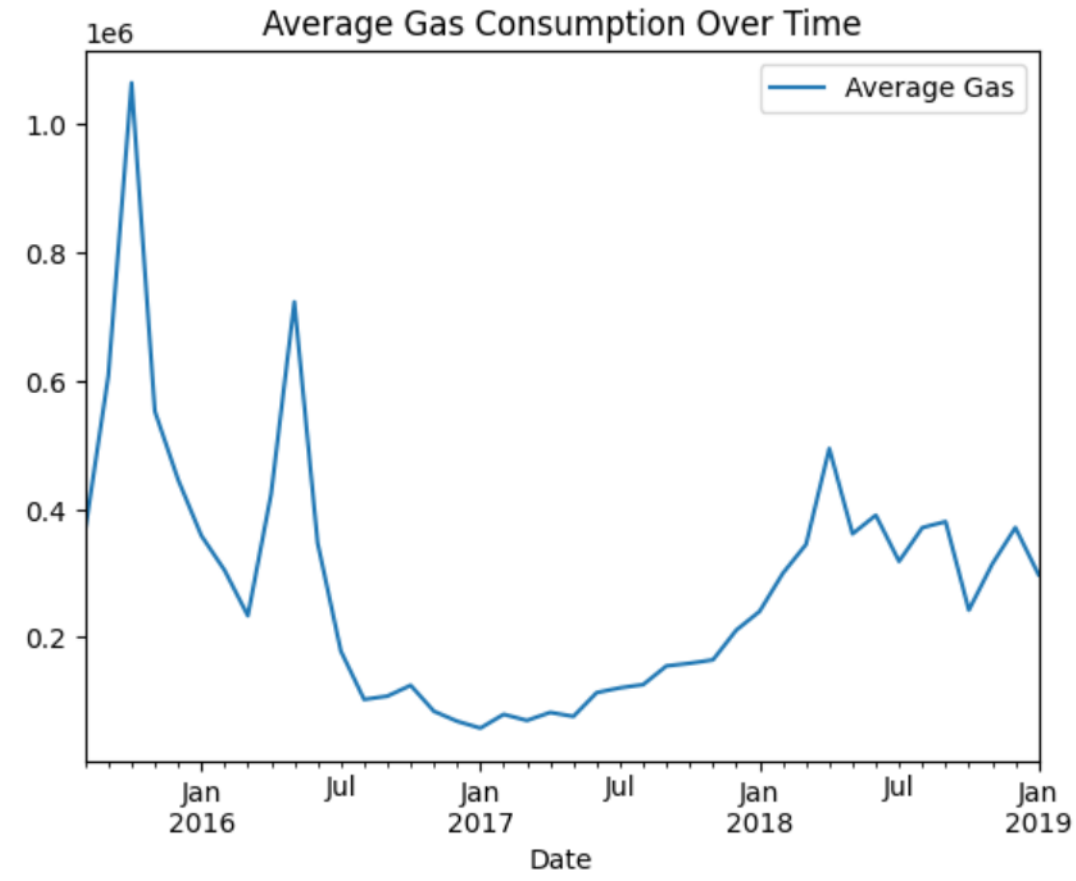


- The most lucrative form of scam has changed through the years, in the initial phase Phishing being the one around in May 2017 till July 2017, after that for a long period of nearly 1 year till June 2018 Fake ICO was the most lucrative form of scam and then it went offline thus losing its position. At the end of our dataset again Phishing is the most lucrative form of scam with that being said, Fake ICO going offline pushes Phishing to the top.
- The ID of the most lucrative form of scam is: 4276.
- The graphs and the ID calculation is done and shown in the python: **Part_D_Scams.ipynb** using the output files from spark: **scams.txt** and **most-lucarative.txt** all available in the zip file.

5 Part D – Gas Guzzlers

- The first step was to filter out any invalid lines from the transactions.csv and contracts.csv files by creating two functions called good_line1 and good_line2. good_line1 function checks if the number of fields in each line is 15, if the address field is a valid string and if the timestamp field is a valid integer for transaction dataset, good_line2 function checks if the number of fields in each line is 6 and if the address field is a valid string for the contract dataset. Any line that fails these checks is filtered out.
- The next step was to read the clean transaction and contract lines from the CSV file using Spark's textFile method, and then apply the two good_line functions to filter out any invalid lines.
- Transaction data is then mapped to a tuple of the form (timestamp, (gas_price, 1)), where timestamp is the year and month of the transaction, gas_price is the price of gas used in the transaction, and 1 is used as a counter to keep track of the number of transactions with that timestamp.
- The resulting RDD is then aggregated using the reduceByKey method, which sums the gas prices and counts for each unique timestamp. This is then directly used to output: gas-price-avg.csv
- Contract data is then mapped to a tuple of the form (address, 'Contract'), where address is the address of the contract and 'Contract' is a string indicating that this is a contract address.
- The clean transactions data is then again mapped to a tuple of the form ((address, timestamp), (gas_used, 1)), where address is the address of the contract associated with the transaction, timestamp is the year and month of the transaction, gas_used is the amount of gas used in the transaction, and 1 is used as a counter to keep track of the number of transactions with that timestamp and address.
- The resulting RDD is then aggregated using the reduceByKey method, which sums the gas used and counts for each unique combination of address and timestamp. This gives us the total gas used for each contract for each month.
- Then we shift the timestamp to values as we need address for joining and it is done by mapping the RDD from previous stage to a tuple of the form (address, (timestamp, gas_used, count)), where address is the address of the contract associated with the transaction, timestamp is the year and month of the transaction, gas_used is the amount of gas used in the transaction, and count is the number of transactions with that timestamp and address.
- The resulting RDD is then joined with the contracts RDD on the address field, giving us a new RDD of the form (timestamp, (gas_used, count)) for each contract.
- Finally, the resulting RDD is aggregated using the reduceByKey method, which sums the gas used and counts for each unique timestamp. This gives us the total gas used for all contracts for each month.
- This is then used to put to the output file: gas-contract.csv
- Graphs showing how gas price has changed over time and graph showing how gas used for contract transactions has changed over time are as follows:





- We can see that over time the contracts have become less complicated and require less gas as compared to the early stages and by looking at the results from Part B we can see that the most popular smart contracts require more gas as they have higher values than the average.

- The python file generating the code is: **Part_D_Gas.ipynb** which is using the output files from spark: **gas-contract.csv** and **gas-price-avg.csv** all of which are in zip.

6 Part D – Data Overheard

- The first step was to filter out any invalid lines from the blocks.csv file by creating a function called good_line. This function checks if the number of fields in each line is 19, and if the sha3_uncles, logs_bloom, transactions_root, state_root, receipts_root fields are a valid string. Any line that fails these checks is filtered out.
- The next step was to read the clean block lines from the CSV file using Spark's textFile method, and then apply the good_line function to filter out any invalid lines.
- Block data is then mapped using ‘Memory’ string as the key and length of all the mentioned fields above as the value, note that a factor of -2 is there to incorporate 0x.
- The resulting RDD is then reduced by reduceByKey, calculating sum of all the lengths for each column.
- The same is then taken as an output as block-data.csv which is then used in Excel to multiply each length by 4 and thus calculating the total memory saved.
- The output is shared as **block-data.csv** in the zip folder.

Name	sha3_uncles	logs_bloom	transactions_root	state_root	receipts_root
Length	448000073	3584000520	448000079	448000072	448000075
Memory saved in Bytes	1792000292	14336002080	1792000316	1792000288	1792000300
Total Memory Saved					21504003276