ANALYSIS OF ETHEREUM TRANSACTIONS AND SMART CONTRACTS

ECS765P - BIG DATA PROCESSING - 2022/23 - SEMESTER 2

INTRODUCTION

Ethereum is a blockchain based distributed computing platform where users may exchange currency (Ether), provide or purchase services (smart contracts), mint their own coinage (tokens), as well as other applications. The Ethereum network is fully decentralized, managed by public-key cryptography, peer-to-peer networking, and proof-of-work to process/verify transactions. A subset of the Ethereum data is provided on the data repository bucket **/data-repository-bkt/ECS765/ethereum-parvulus.** We are provided with three datasets: ethereum transactions, ethereum block data used to acquire mining details, and ethereum contracts used to design smart contracts.

PART A. TIME ANALYSIS

i) Bar plot showing the number of transactions occurring every month between the start and end of the dataset.

Input File:

- transactions.csv

Source Files:

- transactionscount.py

- trans_count_plot.ipynb

Execution commands:

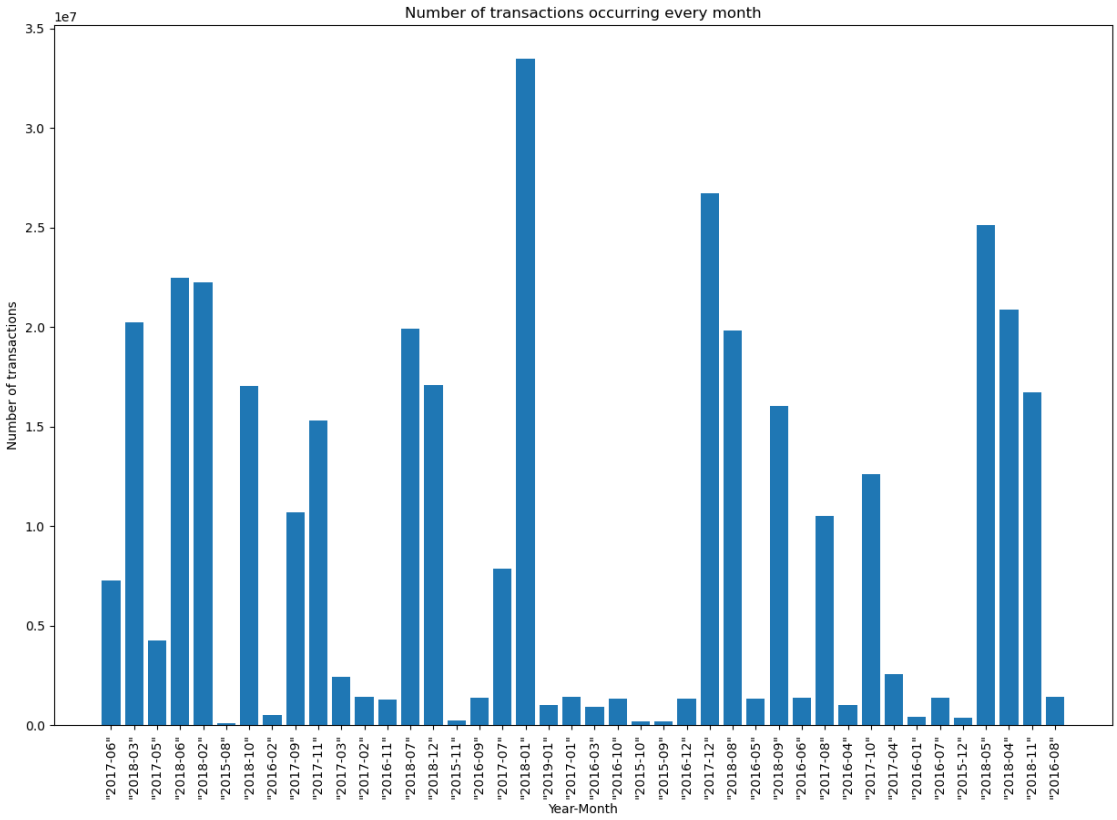
- To start the spark job : **ccc create spark transactionscount.py -s**
- To view the logs of the spark job : **oc logs -f transactionscount-spark-app-driver**
- To access the contents in our bucket : **ccc method bucket ls**
- To save the output to our local directory : **ccc method bucket cp - r bkt:ethereum_30-03-2023_23:21:05/ output**
- This will save an output folder **no_of_transactions.txt** in the local directory.

Methodology:

- Import the necessary libraries : "*pyspark*", "*datetime*", "*boto3*", and "*json*".
- Then, a SparkSession is created and configured with the application name "Ethereum".This session reads the Ethereum transaction data from the S3 container.
- A function *good_lines()* is defined that checks if the input file has exactly 15 fields and if the 12th field (containing the timestamp) is an integer. If both conditions are satisfied, the function returns *True*, otherwise *False*. This varies from file to file based on the number of columns present and we perform necessary validation checks.
- Environment variables are retrieved to get the S3 endpoint URL, access key ID, secret access key, and bucket name. The Hadoop configuration is set with the S3 endpoint URL, access key ID, and secret access key.
- The *transactions.csv* file is read from S3 using *textFile()*, and invalid transactions are filtered out using *filter()* with the *good_lines()* function.
- The *map()* transformation is then applied to the cleaned RDD to extract the month and year information from the timestamp field (12th field in the transaction data) and create a key-value pair where the key is the month and year information and the value is 1. This will be used to count the number of transactions occurring in each month.
- The *reduceByKey()* transformation is applied to the key-value pairs RDD to calculate the total number of transactions occurring in each month. This is done by adding up the values of all the key-value pairs with the same key (i.e., the same month and year).
- The current date and time are obtained using *datetime.now()*. This will be used to create a unique folder name to store the output file in the S3 bucket.

- A *boto3* resource is created to interact with the S3 bucket using the AWS credentials.
- A result object is created with the filename *no_of_transactions.txt* and stored in the S3 bucket with the contents of the *take(100)* (top 100 records) items from the transactions by month RDD, converted to a JSON format.
- The contents of the transactions by month RDD are then printed to the console. Finally, the Spark session is stopped to release the resources.

***Bar plot showing the number of transactions occurring every month with year is as follows**:*



From the above plot, we can see that the maximum number of transactions has occurred in January 2018.

ii) Bar plot showing the average value of transaction in each month between the start and end of the dataset.

Input File:

- transactions.csv

Source Files:

- avgtransactions.py
- avg_trans_plot.ipynb

Execution commands:

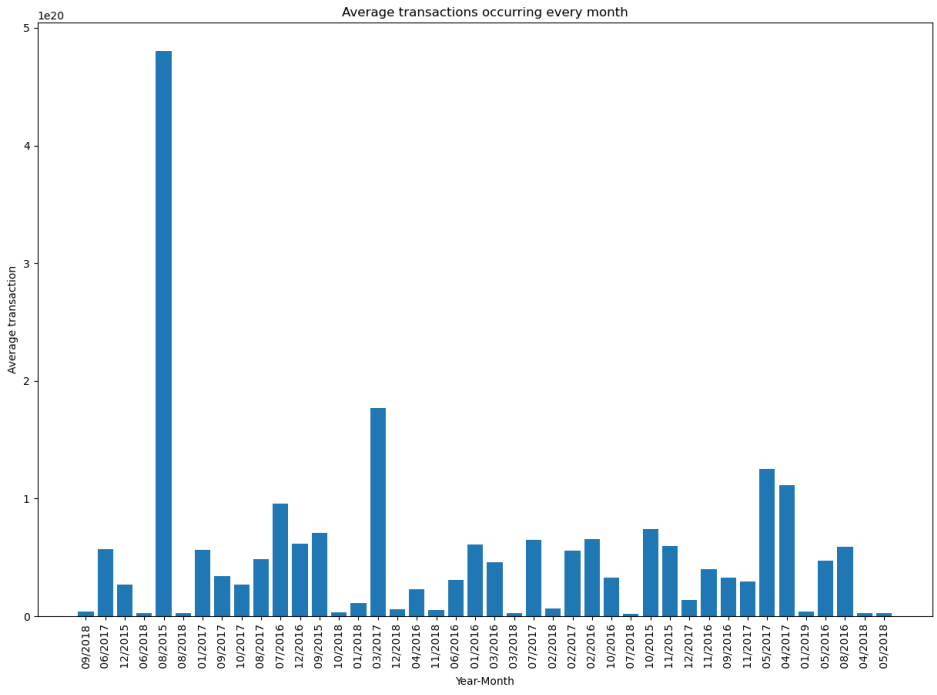
- To start the spark job : **ccc create spark avgtransactions.py -s**
- To view the logs of the spark job : **oc logs -f avgtransactions-spark-app-driver**
- To access the contents in our bucket : **ccc method bucket ls**
- To save the output to our local directory : **ccc method bucket cp - r bkt:ethereum_avg30-03-2023_23:40:05/ output**
- This will save an output folder **transactions_avg.txt** in the local directory.

Methodology:

- Import the necessary libraries : *"sys", "string", "os", "socket", "time", "operator", "boto3", "json",* and "*SparkSession*" from "*pyspark.sql".*
- Then, a SparkSession is created and configured with the application name "Ethereum".This session reads the Ethereum transactions data from the S3 container.
- A function named *good_lines* is defined that checks whether a given transaction line is valid or not. The function checks whether the line has all 15 fields and whether the value and timestamp fields are numeric.
- Environment variables are retrieved to get the S3 endpoint URL, access key ID, secret access key, and bucket name. The Hadoop configuration is set with the S3 endpoint URL, access key ID, and secret access key.

- The *transactions.csv* file is read from S3 using *textFile()*, and invalid transactions are filtered out using *filter()* with the *good_lines()* function.
- The average value of transactions per month is calculated using *map()* to split the transaction line, *reduceByKey()* to aggregate the sum of transaction values and count of transactions per month, and *map()* again to calculate the average transaction value per month. The output is then formatted and saved as a string.
- The current date and time are obtained using *datetime.now()*. This will be used to create a unique folder name to store the output file in the S3 bucket.
- A *boto3* resource is created to interact with the S3 bucket using the AWS credentials.
- A result object is created with the filename *transactions_avg.txt* and stored in the S3 bucket with the contents of the *take(100)* (top 100 records) items from the transactions by month RDD, converted to a JSON format.
- The contents of the transactions by month RDD are then printed to the console. Finally, the Spark session is stopped to release the resources.

***Bar plot showing the average transactions occurring every month with year is as follows**:*



From the above plot, we can see that in the month of August in 2015, the average transaction is maximum.

PART B. TOP TEN MOST POPULAR SERVICES

Top ten smart contracts by total ether received

Input File:

- transactions.csv
- contracts.csv

Source Files:

- popularservices.py

Execution commands:

- To start the spark job : **ccc create spark popularservices.py -s**
- To view the logs of the spark job : **oc logs -f popularservices-spark-app-driver**
- To access the contents in our bucket : **ccc method bucket ls**
- To save the output to our local directory : **ccc method bucket cp - r bkt:ethereum_top_popular_services_30-03-2023_23:58:05/ output**
- This will save an output folder **top_popular_services.txt** in the local directory.

Methodology:

- Import the necessary libraries : "*pyspark*", "*datetime*", "*boto3*", and "*json*".
- Then, a SparkSession is created and configured with the application name "Ethereum".This session reads the Ethereum transaction and contracts data from the S3 container.
- Two functions *good_lines_trans* and *good_lines_contracts* are defined to filter out any bad lines of data from the input files (transactions.csv and contracts.csv). These functions will be used later to ensure that the data being analyzed is clean.

- In *good_lines_trans* the function takes in a line from the transactions.csv file and returns True if the line is valid and False otherwise. A valid line has 15 fields and the 4th field can be converted to an integer.
- In *good_lines_contracts* the function takes in a line from the contracts.csv file and returns True if the line is valid and False otherwise. A valid line has 6 fields.
- Environment variables are retrieved to get the S3 endpoint URL, access key ID, secret access key, and bucket name. The Hadoop configuration is set with the S3 endpoint URL, access key ID, and secret access key.
- The *transactions.csv* and *contracts.csv* files are read from S3 using *textFile()*, and invalid transactions are filtered out using *filter()* with the *good_lines_trans()* and *good_lines_contracts* function.
- The *map* and *reduceByKey* methods are applied to the transactions to group the transactions by service category and sum up the transaction values for each category.
- The *map* method is applied to the contracts to count the number of contracts for each service category.
- The *join* method is used to join the transactions and contracts RDDs together based on the service category.
- The *takeOrdered* method is used to return the top 10 most popular services based on their transaction values.
- The current date and time are obtained using *datetime.now()*. This will be used to create a unique folder name to store the output file in the S3 bucket.
- A *boto3* resource is created to interact with the S3 bucket using the AWS credentials.
- A result object is created with the filename *top_popular_services.txt* and stored in the S3 bucket with the top ten popular services and later, converted to a JSON format.
- The contents of the top popular services are then printed to the console. Finally, the Spark session is stopped to release the resources.

The top ten smart contracts by the ether received is as follows:

ADDRESS	ETHEREUM VALUE
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444	84155363699941767867374641
0x7727e5113d1d161373623e5f49fd568b4f543a9e	45627128512915344587749920
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef	42552989136413198919298969

0xbfc39b6f805a9e40e77291aff27aee3c96915bdd	21104195138093660050000000
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3	155430776352637422547194 09
0xabbb6bebfa05aa13e908eaa492bd7a8343760477	10719485945628946136524680
0x341e790174e3a4d35b65fdc067b6b5634a61caea	8379000751917755624057500
0x58ae42a38d6b33a1e31492b60465fa80da595755	2902709187105736532863818
0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3	1238086114520042000000000
0xe28e72fcf78647adce1f1252f240bbfaebd63bcc	1172426432515823142714582

PART C. TOP TEN MOST ACTIVE MINERS

Top ten miners by the size of the blocks mined

Input File:

- blocks.csv

Source Files:

- topminers.py

Execution commands:

- To start the spark job : **ccc create spark topminers.py -s**
- To view the logs of the spark job : **oc logs -f topminers-spark-app-driver**
- To access the contents in our bucket : **ccc method bucket ls**
- To save the output to our local directory : **ccc method bucket cp - r bkt:ethereum_top_miners_03-04-2023_12:58:05/ output**
- This will save an output folder **top_miners.txt** in the local directory.

Methodology:

- Import the necessary libraries : “*pyspark*”, “*datetime*”, “*boto3*”, and “*json*”.
- Then, a SparkSession is created and configured with the application name "Ethereum".This session reads the Ethereum blocks data from the S3 container.
- Invalid lines are filtered out by using the *filter()* method and passing the *good_lines()* function as an argument. Then, the miner and block size are extracted from the valid lines by using the *map()* method and passing the *feature_extraction()* function as an argument.
- To find the top 10 most active miners, the features RDD is reduced by key (i.e., miner) by using the *reduceByKey()* method and adding up the block sizes. Finally, *takeOrdered()* method is used to get the top 10 miners based on block size.
- The results are uploaded to S3 by using the *boto3.client()* method to create an S3 client and then calling the *put_object()* method to upload the top 10 miners data in a JSON format. The key is created by concatenating the date and time to the file name. The output file is stored in *top_miners.txt* file.
- Finally, the Spark session is stopped using the *stop()* method.

The top ten miners by the size of the blocks mined is as follows:

MINER	BLOCK SIZE
0xea674fdde714fd979de3edf0f56aa9716b898ec8	17453393724
0x829bd824b016326a401d083b33d092293333a830	12310472526
0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c	8825710065
0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5	8451574409
0xb2930b35844a230f00e51431acae96fe543a0347	6614130661
0x2a65aca4d5fc5b5c859090a6c34d164135398226	3173096011
0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb	1152847020
0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01	1134151226
0x1e9939daaad6924ad004c2560e90804164900341	1080436358
0x61c808d82a3ac53231750dadc13c777b59310bd9	692942577

PART D. DATA EXPLORATION

i) Scam Analysis : Popular Scams

Identifying the most lucrative scams and analyzing the ether change over time.

Input File:

- transactions.csv
- scams.csv

Source Files:

- popularscams.py
- ether_vs_time.ipynb

Execution commands:

- To start the spark job : **ccc create spark popularscams.py -s**
- To view the logs of the spark job : **oc logs -f popularscams-spark-app-driver**
- To access the contents in our bucket : **ccc method bucket ls**
- To save the output to the local directory : **ccc method bucket cp - r bkt:ethereum_most_lucrative_scams_05-04-2023_21:58:05/ output** and **ccc method bucket cp - r bkt:ethereum_ether_vs_time_05-04-2023_21:58:05/ output**
- This will save an output folder **most_lucrative_scams.txt** and **ether_vs_time.txt** in the local directory.

Methodology:

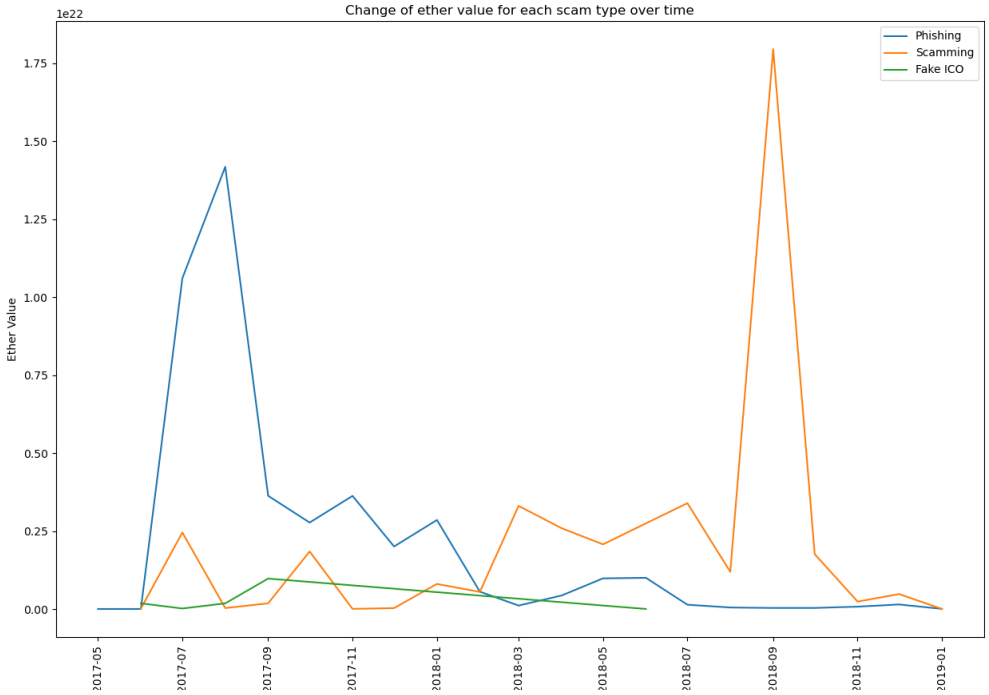
- Import the necessary libraries : "*pyspark*", "*datetime*", "*boto3*", and "*json*".
- Then, a SparkSession is created and configured with the application name "Ethereum".This session reads the Ethereum transaction data from the S3 container.

- The *good_lines_trans()* function is defined, which takes a *line* as input and returns a boolean value based on whether the input is a valid Ethereum transaction or not. The function checks whether the input has exactly 15 fields, the 12th field is an integer, the 7th field is a string, and the 8th field is a float.
- The *good_lines_scams()* function is defined, which takes a *line* as input and returns a boolean value based on whether the input is a valid Ethereum scam or not. The function checks whether the input has exactly 8 fields, the 1st field is an integer, the 5th field is a string, and the 7th field is a string.
- Environment variables are retrieved to get the S3 endpoint URL, access key ID, secret access key, and bucket name. The Hadoop configuration is set with the S3 endpoint URL, access key ID, and secret access key.
- The *transactions.csv* and *scams.csv* files are read from S3 using *textFile()*, and invalid transactions are filtered out using *filter()* with the *good_lines_trans()* and *good_lines_scams* function.
- For finding the most lucrative scams, *scam_mapping_1* and *trans_mapping_1* RDDs are created by mapping the scam and transaction data to a common key, which is the Ethereum address. These RDDs are then joined on the Ethereum address.
- The resulting RDD contains the Ethereum address, scam name, and transaction amount. The *popular_scams* RDD is then created by summing up the transaction amounts for each scam name and sorting them in descending order to find the top 10 most lucrative scams.
- For finding the time of transactions for each scam, *scam_mapping_2* and *trans_mapping_2* RDDs are created by mapping the scam and transaction data to a common key, which is again the Ethereum address. These RDDs are then joined on the Ethereum address.
- The resulting RDD contains the Ethereum address, scam name, month/year of transaction, and transaction amount. The *ethertime* RDD is then created by summing up the transaction amounts for each scam name and month/year of transaction.
- Finally, the results are written to a text file in the S3 bucket using *my_result_object.put()*. The top 10 most lucrative scams are written to *ethereum_most_lucrative_scams.txt* file, and the time of transactions for each scam is written to *ethereum_ether_vs_time.txt* file.
- The PySpark session is then stopped using *spark.stop()* function.

The top ten most lucrative scam is as follows:

SCAM ID	SCAM TYPE	ETHER VALUE
5622	Scamming	1.6709083588072934e+22
2135	Phishing	6.583972305381557e+21
90	Phishing	5.972589629102418e+21
2258	Phishing	3.462807524703739e+21
2137	Phishing	3.389914242537183e+21
2132	Scamming	2.428074787748574e+21
88	Phishing	2.067750892013527e+21
2358	Scamming	1.8351766714814872e+21
2556	Phishing	1.803046574264181e+21
1200	Phishing	1.6305774191330897e+21

The change of ether value for each scam type over time is as follows:



From the above plot, we can see that scamming has been the most common type of scam throughout the period covered by the data. It peaked in March 2018, with a value of 3.31 x 10^21, and remained high until July 2018. After that, it decreased, but still had some spikes in October and December 2018. Phishing has been the second most common type of scam. It peaked in July 2017, with a value of 1.42 x 10^22, and had other spikes in September and December 2017, and March, April, and August 2018. Fake ICOs were the least common type of scam, with only two instances occurring in June and September 2017.

ii) Miscellaneous Analysis : Gas guzzler

Analysis of gas price and gas used for contract transactions over time.

Input File:

- transactions.csv
- contracts.csv

Source Files:

- gasguzzler.py
- Avg_gas_used.ipynb
- Avg_gas_price.ipynb

Execution commands:

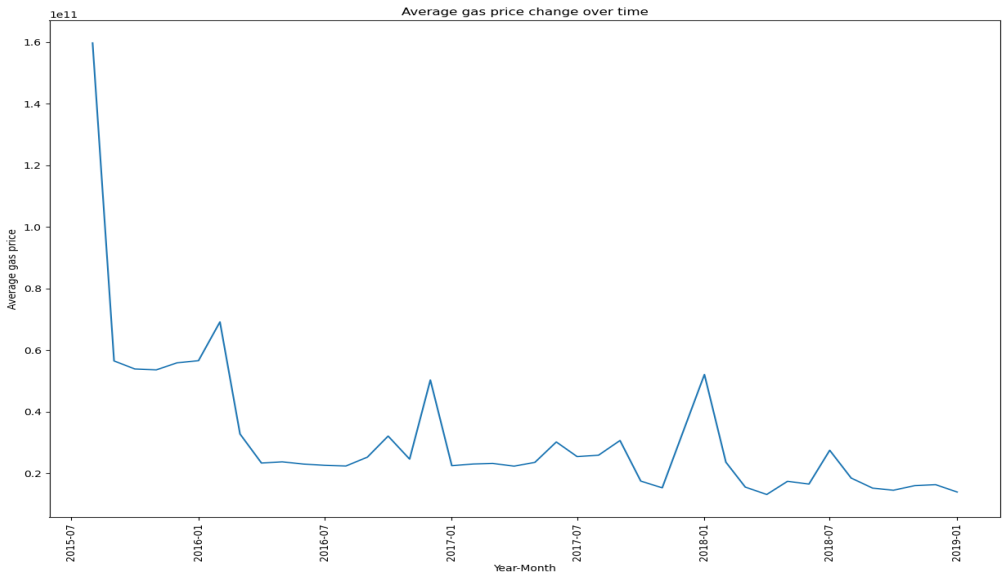
- To start the spark job : **ccc create spark gasguzzler.py -s**
- To view the logs of the spark job : **oc logs -f gasguzzler-spark-app-driver**
- To access the contents in our bucket : **ccc method bucket ls**
- To save the output to the local directory : **ccc method bucket cp - r bkt:ethereum_avg_gas_price_03-04-2023_21:58:05/ output** and **ccc method bucket cp - r bkt:ethereum_avg_gas_used_03-04-2023_21:58:05/ output**
- This will save an output folder **avg_gasprice.txt** and **avg_gasused.txt** in the local directory.

Methodology:

- Import the necessary libraries : “*pyspark”, “datetime”, “boto3”,* and *“json*”.
- Then, a SparkSession is created and configured with the application name "Ethereum".This session reads the Ethereum transaction data from the S3 container.
- The function *good_lines_trans* is defined to check the formatting of each line in the transaction CSV file. It splits each line by comma and checks if the number of fields is 15. If not, it returns False. If the number of fields is correct, it tries to convert the 10th and 12th fields to float. If there is an error in the conversion, it returns False. Otherwise, it returns True.
- The function *good_lines_contracts* is defined to check the formatting of each line in the contract CSV file. It splits each line by comma and checks if the number of fields is 6. If not, it returns False. Otherwise, it returns True.
- Environment variables are retrieved to get the S3 endpoint URL, access key ID, secret access key, and bucket name. The Hadoop configuration is set with the S3 endpoint URL, access key ID, and secret access key.
- The *transactions.csv* and *contracts.csv* files are read from S3 using *textFile(),* and invalid transactions are filtered out using *filter()* with the *good_lines_trans()* and *good_lines_contracts* function.
- The *gas_prices* is created by mapping each line in the *transactions* to a key-value pair where the key is the month and year of the transaction (obtained from the 12th field of the line converted to a human-readable format using *time.gmtime* and *time.strftime* functions) and the value is a tuple containing the gas price (obtained from the 10th field of the line converted to float) and 1 (to later compute the sum and count of gas prices).
- The *gas_prices_reduced* is created by reducing the *gas_prices* by key (i.e., month and year) using *reduceByKey* method. The values for each key are aggregated by summing the gas prices and counts. The result is a key-value pair where the key is the month and year and the value is a tuple containing the sum of gas prices and the count of gas prices.
- The *average_gas_price* is created by mapping each key-value pair in the *gas_prices_reduced* to a new key-value pair where the key is the month and year and the value is the average gas price (i.e., the sum of gas prices divided by the count of gas prices). The *sortByKey* method is used to sort the by key in ascending order.
- The *average_gas_used* is created by mapping the transactions to a tuple with the contract address and a tuple with the month and year and the gas used. The contracts are then mapped to a tuple with the contract address and the value 1.

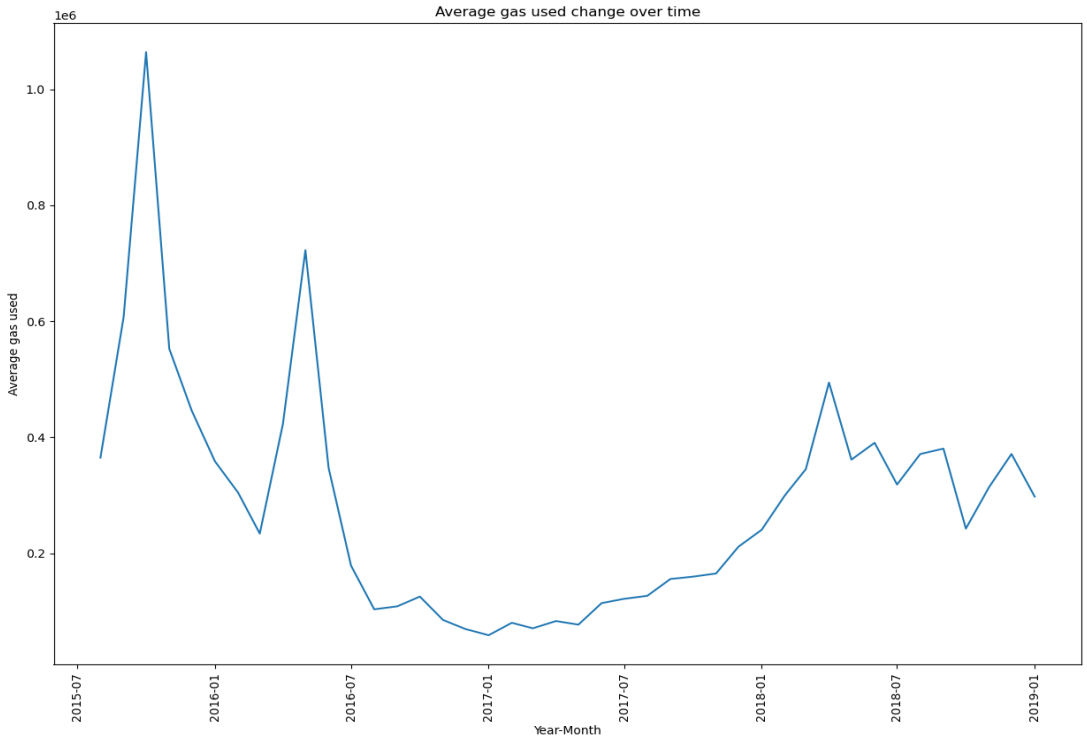
- The *gas_used* is then joined with the contracts on the contract address to create a new *joined_data*, where each key-value pair consists of a contract address and a tuple containing the month and year of the transaction and the value of 1.
- The *gas_used_reduced* is created by mapping each key-value pair in the *joined_data* to a new key-value pair where the key is the month and year and the value is a tuple containing the sum of gas used and the count of transactions for that month and year.
- The map function is used to calculate the average gas used for each month and year, and the result is stored in the *average_gas_used*. This RDD is also sorted by key in ascending order.
- The datetime module is used to get the current date and time, which is formatted as a string and stored in the *date_time* variable.
- The boto3 library is used to create a resource object for S3, using the endpoint URL, access key ID, and secret access key from the environment variables. The S3 bucket name is also obtained from the environment variables.
- Two result objects are created using the *my_bucket_resource* Object method, one for the *average_gas_price* and one for the *average_gas_used*. The put method is used to write the contents of the RDD to a text file in the S3 bucket, with the file name containing the current date and time.
- The Spark session is stopped using the stop method.

***Plot showing the change in gas price over time is as follows**:*



From the above plot, we can see that there is a clear upward trend in the gas prices from 2015 to 2018, with occasional fluctuations and drops. The data also shows a significant increase in gas prices in mid-2017. Furthermore, there appears to be seasonality in the gas prices, with generally higher prices observed during the months of June to August.

Plot showing the change in gas used for contract transactions over time is as follows:



From the above plot, we can see that the gas used per transaction varied widely over time, with some months showing a significant increase in gas usage compared to others. For example, we can see that gas usage was relatively low in the first half of 2016, but increased significantly in the latter half of the year. There were also spikes in gas usage in October 2015 and October 2016, likely due to increased activity on the network during those months.