PART A – Time Analysis

1. **Objective** - The aim here is to create a bar plot that shows the number of transactions occurring every month between the start and end of the dataset.

Relevant Source Code Files - transaction.py, transaction.ipynb

Output Files - transactions_total_05-04-2023_14_29_26.txt is the output for transaction.py

Execution Command - ccc create spark transaction.py -s -e 10

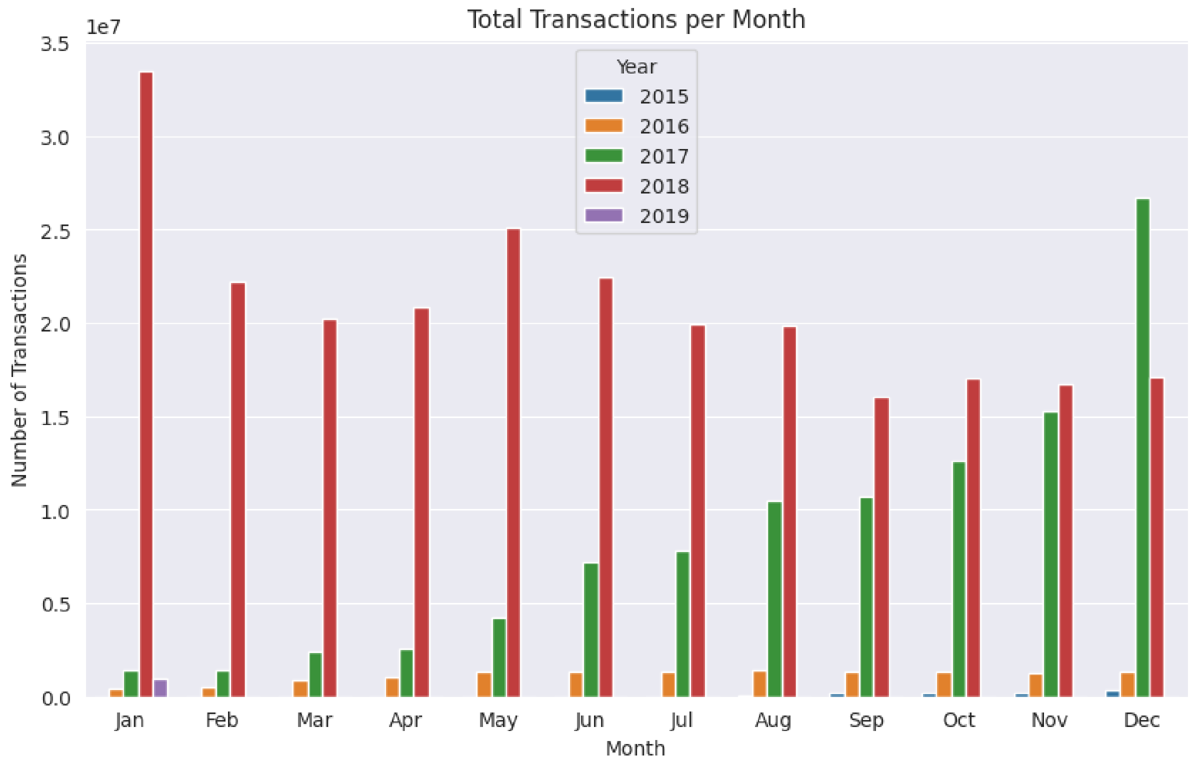
Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-9e5ddcafa8c7466f971f0141e71695ac	Ethereum	2023-04-05 15:29:10	2023-04-05 15:34:49	5.6 min	ec22387	2023-04-05 15:34:50

Command Used to Copy Output File from Bucket to Storage - ccc method bucket cp bkt: transactions_total_05-04-2023_14_29_26.txt ~/teaching_material/ECS765P/BigData-Spark-Project/A.TimeAnalysis/

Explanation - Our goal was to analyze a dataset of Ethereum transactions and determine the monthly transaction count. We started by creating a Spark session and defining functions to filter out invalid data and extract the month and year from each transaction timestamp. We then loaded the dataset from an S3 bucket and applied our filter and processing functions using Spark transformations. Subsequently, we aggregated the transaction counts by month and year using the reduceByKey function. In the end, we saved our results to an S3 bucket as a text file and printed them to the console. We then use Seaborn to make a bar plot visualization showing the monthly transaction count over the given period.

Bar Plot (Number of Transactions vs Month)



2. **Objective** - The aim here is to create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

Relevant Source Code Files - average.py, average.ipynb

Output Files - avg_transactions_05-04-2023_14_41_08.txt is the output for average.py

Execution Command - ccc create spark average.py -s -e 10

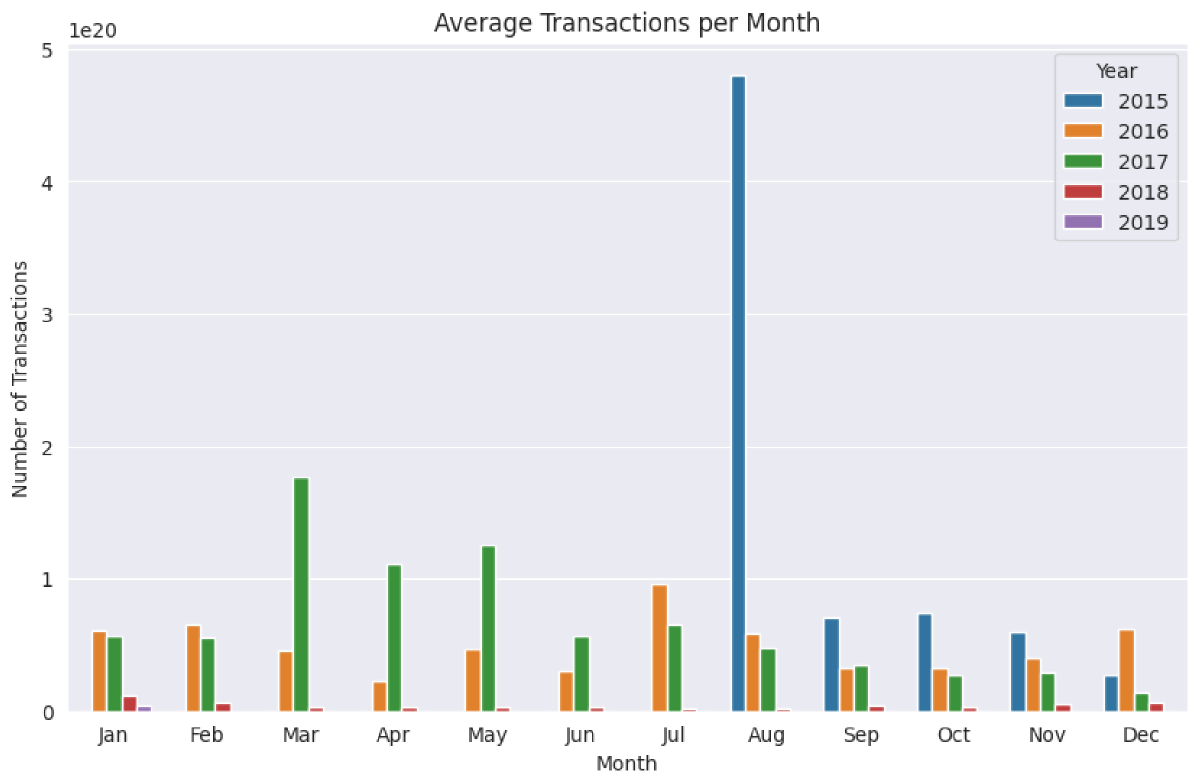
Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-c47190541d384d44a227725afa166f4a	Ethereum	2023-04-05 15:40:52	2023-04-05 15:47:48	6.9 min	ec22387	2023-04-05 15:47:49

Command Used to Copy Output File from Bucket to Storage - ccc method bucket cp bkt: avg_transactions_05-04-2023_14_41_08.txt ~/teaching_material/ECS765P/BigData-Spark-Project/A.TimeAnalysis/

Explanation - Our aim is to calculate the average transaction value for each month in the given Ethereum dataset. We start by defining a function called check_transaction that verifies if each line in the dataset is valid by checking if it has the correct number of fields and if the transaction value and timestamp are in the correct data types. Then, we create a mapping function that processes each valid line by extracting the transaction value and converting the timestamp to a month/year format. Next, we use the reduceByKey function to aggregate the transaction values and their counts for each month/year. After that, we use the function calculate_average that we defined to compute the average transaction value for each month/year by dividing the total value by the count. At last, we apply this function using the map operation and convert the results into strings for storage. We use Seaborn then, just like we did in task 1, to create bar plot visualization.

Bar Plot (Number of Transactions vs Month)



PART B – Top Ten Most Popular Services

Objective - The goal is to evaluate the top 10 smart contracts by total Ether received.

Relevant Source Code Files - top10services.py, top10services.ipynb

Output Files - top10services_05-04-2023_15_02_50.txt is the output for top10services.py

Execution Command - ccc create spark top10services.py -s -e -15

Command Used to Copy Output File from Bucket to Storage - ccc method bucket cp bkt: top10services_05-04-2023_15_02_50.txt ~/teaching_material/ECS765P/BigData-Spark-Project/B.Top10PopularServices/

Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-0a7260ef892e48819da3b47527d8c9d2	Ethereum	2023-04-05 15:55:45	2023-04-05 16:02:50	7.1 min	ec22387	2023-04-05 16:02:52

Explanation - Here's a detailed step by step overview of what we have accomplished in the code written in top10services.py,

1. **Set up a Spark session**: We created a Spark session with the application name "Ethereum" using the SparkSession builder.
2. **Configure S3 access**: We set up environment variables for accessing the S3 bucket and configured Hadoop to work with S3 using the provided access key, secret key, and endpoint URL.
3. **Load data from S3**: We read the transactions and contracts data from the S3 bucket using the textFile method.
4. **Filter valid data**: We filtered valid transactions and contracts by checking if the number of fields in each line is correct and if specific fields contain valid data (e.g., checking if the transaction value is a digit).
5. **Process transactions and contracts**: We created tuples for transactions and contracts containing the relevant data (addresses and values). We then aggregated transaction values for each address and merged the aggregated transaction values with contract addresses.
6. **Find the top 10 smart contracts**: We extracted the address and corresponding transaction value from the merged_transactions_contracts RDD and retrieved the top 10 smart contracts with the highest transaction values.
7. **Set up S3 bucket resource**: We set up an S3 bucket resource using the boto3 library with the provided endpoint URL, access key, and secret key.
8. **Save results to S3**: We formatted the current date and time and used it to construct an output file name. Then, we saved the top 10 smart contracts in the output file to the S3 bucket.
9. **Stop the Spark session**: Finally, we stopped the Spark session using the stop method.

Then we format the output file “top10services_02-04-2023_14:15:52.txt” in top10services.ipynb to get our result in a nicer way.

```
Rank    Address                                       Ethereum Value
----    -------                                       --------------
1       0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444    84155363699941767867374641
2       0x7727e5113d1d161373623e5f49fd568b4f543a9e    45627128512915344587749920
3       0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef    42552989136413198919298969
4       0xbfc39b6f805a9e40e77291aff27aee3c96915bdd    21104195138093660050000000
5       0xe94b04a0fed112f3664e45adb2b8915693dd5ff3    15543077635263742254719409
6       0xabbb6bebfa05aa13e908eaa492bd7a8343760477    10719485945628946136524680
7       0x341e790174e3a4d35b65fdc067b6b5634a61caea    8379000751917755624057500
8       0x58ae42a38d6b33a1e31492b60465fa80da595755    2902709187105736532863818
9       0xc7c7f6660102e9a1fee1390df5c76ea5a5572ed3    1238086114520042000000000
10      0xe28e72fcf78647adce1f1252f240bbfaebd63bcc    1172426432515823142714582
```

PART C – Top Ten Most Active Miners

Objective - The goal is to find the top 10 miners by the size of the blocks mined.

Relevant Source Code Files - top10miners.py, top10miners.ipynb

Output Files - top10miners_05-04-2023_15_08_15.txt is the output for top10miners.py

Execution Command - ccc create spark top10miners.py -s -e -15

Command Used to Copy Output File from Bucket to Storage - ccc method bucket cp bkt: top10miners_05-04-2023_15_08_15.txt ~/teaching_material/ECS765P/BigData-Spark-Project/C.Top10ActiveMiners/

Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-6589f29ff584450eafafd4ea4f29ea07	Ethereum	2023-04-05 16:07:44	2023-04-05 16:08:15	32 s	ec22387	2023-04-05 16:08:16

Explanation - We then filter the data to keep only the valid blocks, which have 19 fields and do not contain the header row. This is achieved using a lambda function that checks the length of the fields and whether the second field ('hash') is not equal to the header value.

Next, we use another lambda function to extract the necessary information from each valid block, specifically the miner's address and the block's size. The lambda function returns a tuple containing the miner's address and the size of the block.

Once we have the required information, we perform a reduceByKey operation on the tuples to aggregate the block sizes for each miner. This operation sums the block sizes for each miner, effectively calculating the total size of blocks mined by each miner.

After aggregating the block sizes, we use the takeOrdered function to retrieve the top 10 miners based on the total block size they have mined. This function sorts the miners in descending order of block size and returns the top 10 miners as a list of tuples, where each tuple contains the miner's address and the corresponding total block size.

Then we format contents of “top10miners_02-04-2023_16_08_38” in top10miners.ipynb to get a nicer output.

```
Rank    Miner                                         Block Size
----    -------                                       -------------
1       0xea674fdde714fd979de3edf0f56aa9716b898ec8   17453393724
2       0x829bd824b016326a401d083b33d092293333a830   12310472526
3       0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c   8825710065
4       0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5   8451574409
5       0xb2930b35844a230f00e51431acae96fe543a0347   6614130661
6       0x2a65aca4d5fc5b5c859090a6c34d164135398226   3173096011
7       0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb   1152847020
8       0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01   1134151226
9       0x1e9939daaad6924ad004c2560e90804164900341   1080436358
10      0x61c808d82a3ac53231750dadc13c777b59310bd9   692942577
```

PART D – Data Exploration

1. Scam Analysis - Popular Scams

Objective:

We achieve two main objectives here:

- Identify the top 10 most lucrative scams.
- Calculate the total Ether received for each scam category in each month/year.

Relevant Source Code Files – scams.py, scams_output.ipynb, visualization.ipynb

Output Files - popular_scams_05-04-2023_15_37_10.txt and ether_against_time_05-04-2023_15_37_10.txt are the output files for scams.py (same copy commands used to copy from bucket).

Execution Command - ccc create spark scams.py -s -e 20

Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-c43e9d29e2514fb7bc93d2b3aa69f70e	Ethereum	2023-04-05 16:30:03	2023-04-05 16:44:33	14 min	ec22387	2023-04-05 16:44:35

Explanation:

To achieve our objectives, the code (scams.py) follows these steps:

1. Define the process_json_to_csv_dict function, which converts JSON data to a list of comma-separated strings in CSV format.
2. Create a Spark session.
3. Set up environment variables and configure Hadoop to access the S3 bucket.
4. Read transaction data in CSV format and scam data in JSON format from the S3 bucket.
5. Convert the JSON data to a dictionary and process it using the process_json_to_csv_dict function to get CSV-formatted data.
6. Create RDDs from the transaction and scam data.
7. Filter out invalid transactions and scams.
8. Map scams to their address and ID/category tuples.
9. Map transactions to their address and Ether received.
10. Join transactions and scams on the address.
11. Map joined data to the scam ID/category and Ether received.
12. Calculate the total Ether received for each scam ID/category.
13. Get the top 15 most lucrative scams.
14. Map transactions to their address and month/year with Ether received.
15. Join transactions with scams on the address.
16. Map joined data to the month/year and scam category with Ether received.
17. Calculate the total Ether received for each scam category in each month/year.
18. Set up the S3 bucket resource for uploading results.
19. Get the current date and time, and construct the output file names with the date and time.
20. Upload the top 15 most lucrative scams to the S3 bucket.
21. Upload the Ether received by scam category and month/year to the S3 bucket.
22. Stop the Spark session.

We format the output file "popular_scams_05-04-2023_15_37_10.txt" in scams_output.ipynb for a nicer output. This is done by first reading the file and then unpacking list of lists. Later, it is stored into a dataframe. The output is as follows,

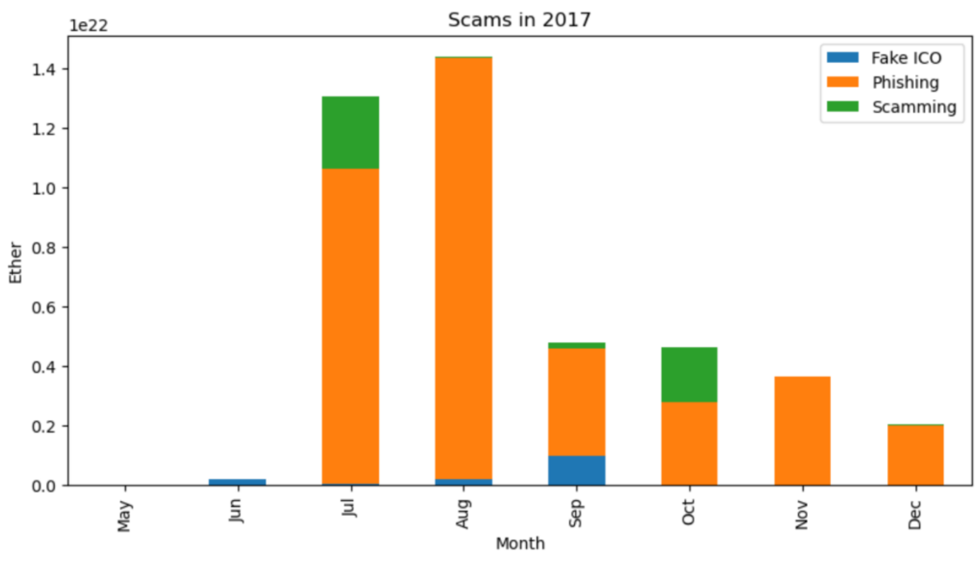
	scam_id	scam_type	ether
0	5622	Scamming	1.670908e+22

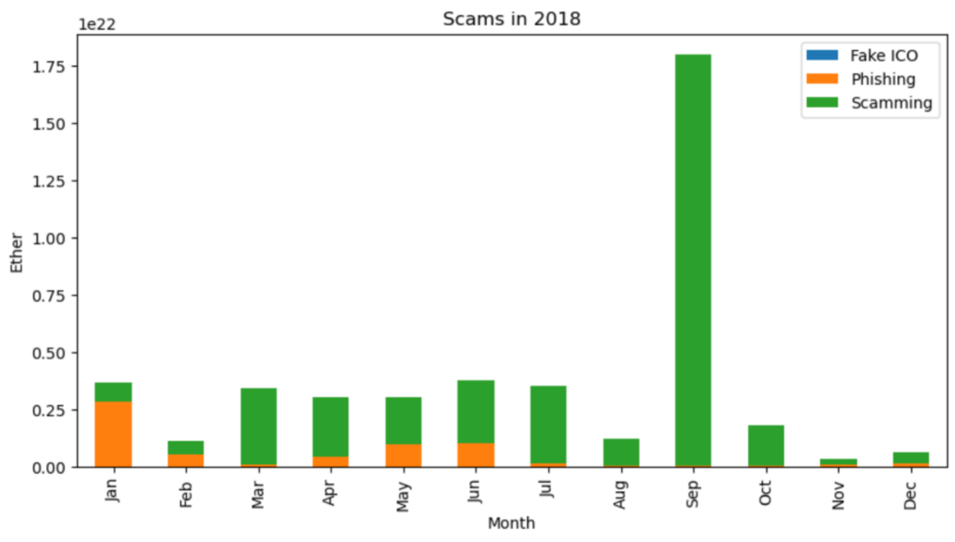
	scam_id	scam_type	ether
1	2135	Phishing	6.583972e+21
2	90	Phishing	5.972590e+21
3	2258	Phishing	3.462808e+21
4	2137	Phishing	3.389914e+21
5	2132	Scamming	2.428075e+21
6	88	Phishing	2.067751e+21
7	2358	Scamming	1.835177e+21
8	2556	Phishing	1.803047e+21
9	1200	Phishing	1.630577e+21
10	2181	Phishing	1.163904e+21
11	41	Fake ICO	1.151303e+21
12	5820	Scamming	1.133973e+21
13	86	Phishing	8.944561e+20
14	2193	Phishing	8.827100e+20

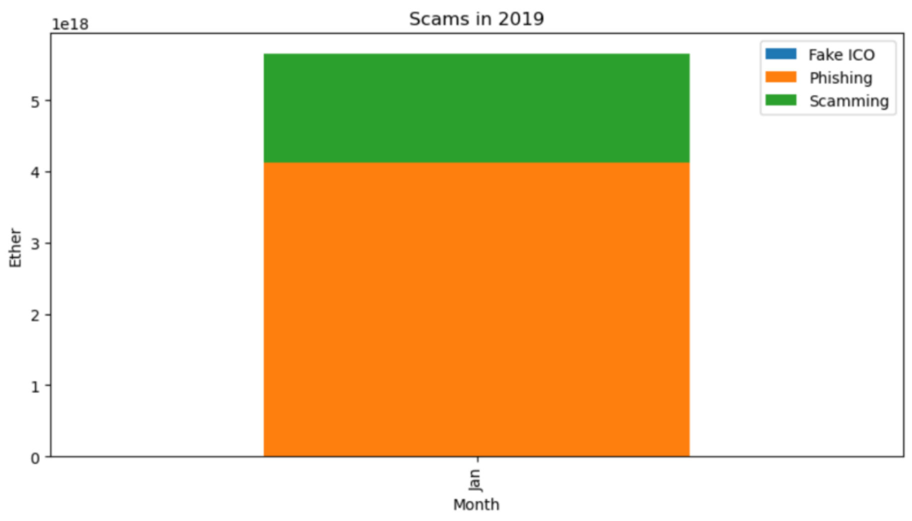
In visualization.ipynb we

1. Read the text file: We start by reading the text file " ether_against_time_05-04-2023_15_37_10.txt" and storing the contents as a string in the variable "data_str".
2. Convert the string to a list of lists: We convert the string into a list of lists using the "literal_eval" function from the "ast" module.
3. Preprocess data and create a DataFrame: We extract the relevant information from each item in the list of lists and store it as a row in a pandas DataFrame.
4. Convert date column to datetime format: We convert the "date" column in the DataFrame to the datetime format using the "pd.to_datetime()" function.
5. Extract year and month from date column: We extract the year and month from the "date" column and create two new columns in the DataFrame.
6. Create a pivot table with year and month as index and scam_type as columns: We create a pivot table with the "year" and "month" columns as the index and "scam_type" column as the columns in the pivot table. We use the "sum" aggregation function to calculate the total "ether" for each month and scam type.
7. Create a bar plot with the x-axis labeled as "Month-Year" by converting the "year" and "month" columns to a single column of datetime objects. We also assign this new column to the "month-year" key in the DataFrame. We then use this new column for the x-axis in the bar plot. The y-axis is labeled as "Ether" and the hue is set to "scam_type". We use a bright color palette for better visualization. Finally, we set the title of the plot to "Scams over time".

Here's the output,







From the visualizations, we can infer that "Phishing" was most profitable in August 2017. And, "Scamming" was most profitable in September 2018. Finally, "Fake ICO" was the most lucrative scam in September 2017.

2. Miscellaneous Analysis - Gas Guzzlers

Objective - The goal is to analyse how gas price and gas used have changed over time.

Relevant Source Code Files - gasguzzlers.py, gas_visualization.ipynb

Output Files - avg_gas_price_05-04-2023_16_10_39.txt and avg_gas_used_05-04-2023_16_10_39.txt are the output files for gasguzzlers.py (same copy commands used to copy from bucket).

Execution Command - ccc create spark gasguzzlers.py -s -e 20

Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-b71d04ee99154eda8e0f47b405914790	Ethereum	2023-04-05 16:58:24	2023-04-05 17:11:03	13 min	ec22387	2023-04-05 17:11:06

Explanation:

The code in gasguzzlers.py accomplishes the following,

1. Filter valid transactions and contracts:
   - Remove transactions with incorrect format, missing values, or non-numeric gas prices and timestamps
   - Remove contracts with incorrect format or missing values
2. Calculate average gas price per month using transaction data:
   - Extract the month-year and gas price for each transaction
   - Sum the gas prices for each month-year and count the number of transactions
   - Calculate the average gas price by dividing the sum by the count for each month-year
3. Identify contract transactions by joining transactions and contracts datasets:
   - Extract transaction hash, timestamp, and gas used from the transactions dataset
   - Extract contract address from the contracts dataset
   - Perform an inner join on transaction hash and contract address, retaining only contract transactions
4. Calculate average gas used for contract transactions per month:
   - Extract the month-year and gas used for each contract transaction
   - Sum the gas used for each month-year and count the number of contract transactions
   - Calculate the average gas used by dividing the sum by the count for each month-year
5. Sort the results by month-year and store them in the S3 bucket:
   - Sort the average gas price and average gas used results by month-year in ascending order
   - Store the sorted results in two separate files avg_gas_price_05-04-2023_16_10_39.txt and avg_gas_used_05-04-2023_16_10_39.txt in the S3 bucket

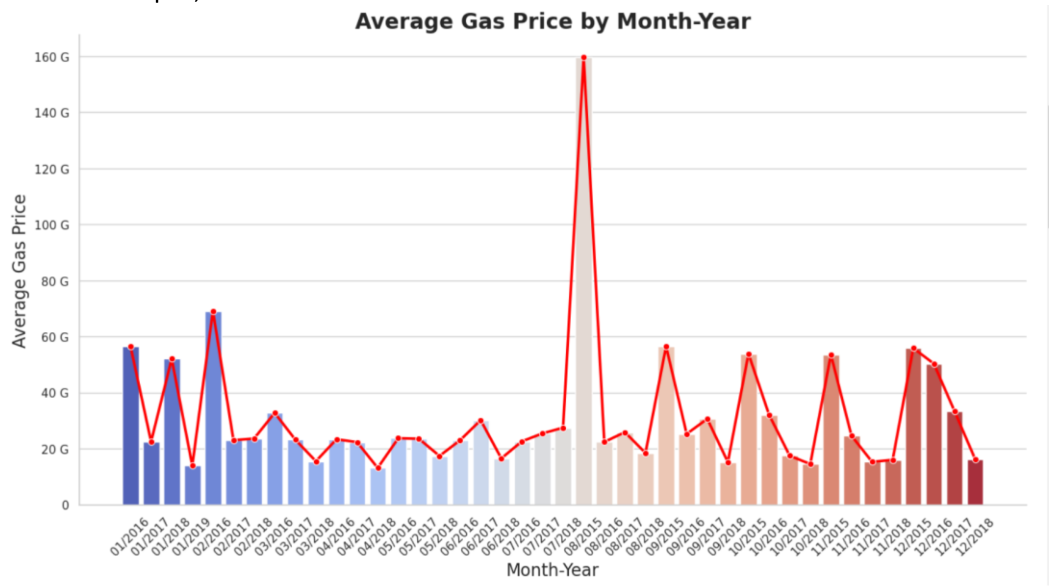
In gas_visualization.ipynb we,

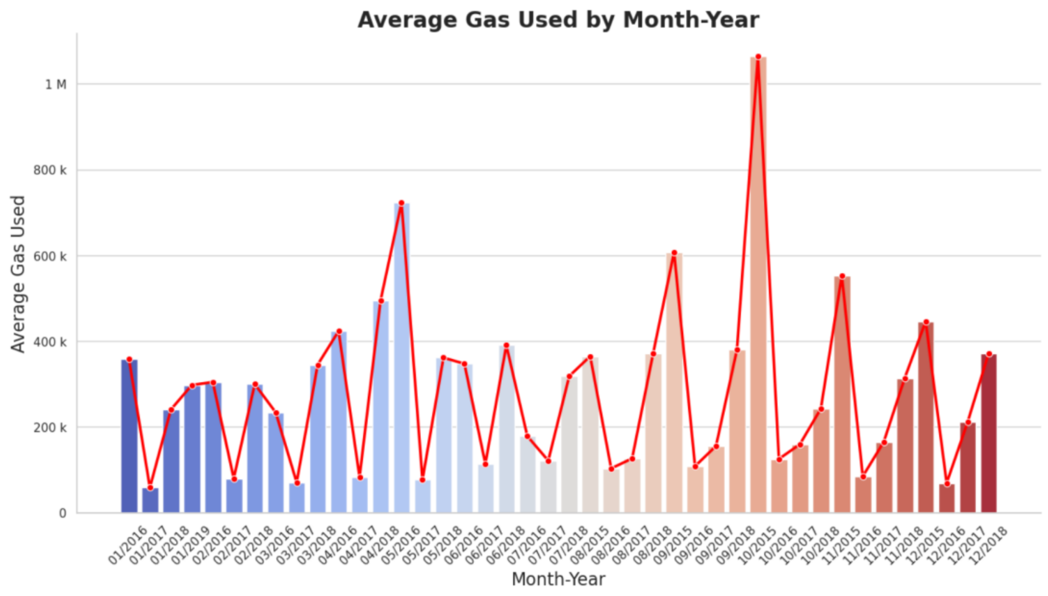
1. Read data from files:
   - Open and read the avg_gas_price_05-04-2023_16_10_39.txt and avg_gas_used_05-04-2023_16_10_39.txt files
   - Convert the content of the files to Python lists
2. Create pandas dataframes:
   - Convert the lists to pandas dataframes with appropriate column names
   - Convert the 'Avg Gas Price' and 'Avg Gas Used' columns to float data type
3. Set up Seaborn and Matplotlib:
   - Use Seaborn's whitegrid theme for styling
   - Create a figure with a specified size using Matplotlib
4. Create bar plots for average gas price and average gas used:
   - Use Seaborn's barplot function with custom color palettes
   - Set the x-axis and y-axis labels and customize the title, font size, and font weight

- Rotate the x-axis labels for better readability

5. **Add line plots to the existing bar plots:**
   - Use Seaborn's lineplot function with red color and circle markers
   - Customize the line width
6. **Customize y-axis tick format:**
   - Use Matplotlib's ticker to format the y-axis ticks with engineering notation
7. **Remove top and right spines:**
   - Use Seaborn's despine function to remove the top and right spines for a cleaner appearance
8. **Display the plots:**
   - Use Matplotlib's show function to display the plots on the screen

Here's the output,





From the average gas price data we gather the following,

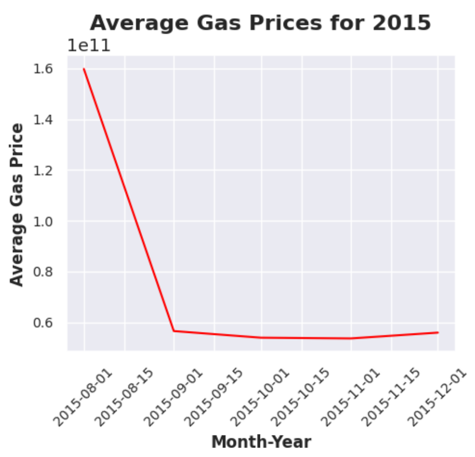
- The given average gas price data shows fluctuations over time, from 2015 to 2018.
- In August 2015, the average gas price was exceptionally high. However, this value decreased significantly in subsequent months, showing a downward trend.
- Throughout 2016, the average gas price fluctuated within a broad range, with the highest gas price occurring in February and the lowest in April.

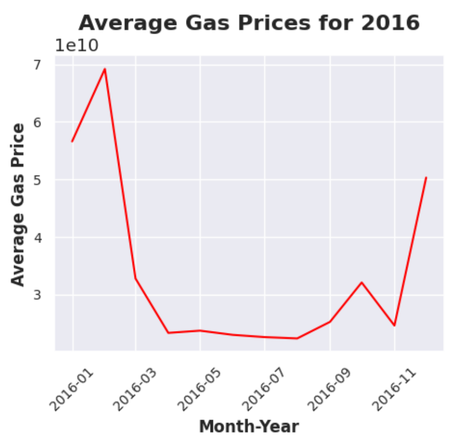
- In 2017, the average gas price continued to vary but remained within a narrower range. There were two peaks around June and September, while the lowest happened in November.

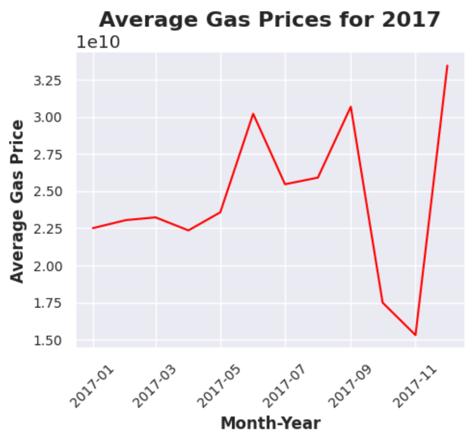
- The data for 2018 shows a general downward trend in average gas prices. The year started with a relatively high average gas price in January, and then it gradually decreased to reach its lowest point in December.
- In conclusion, the average gas price shows fluctuations over the years with an overall downward trend from 2015 to 2018. The data indicates that gas prices on Ethereum have not remained constant and have experienced significant variations during this period.

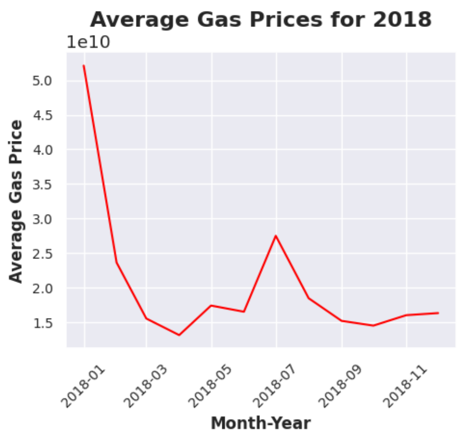
Similarly from the average gas used data we infer the following,

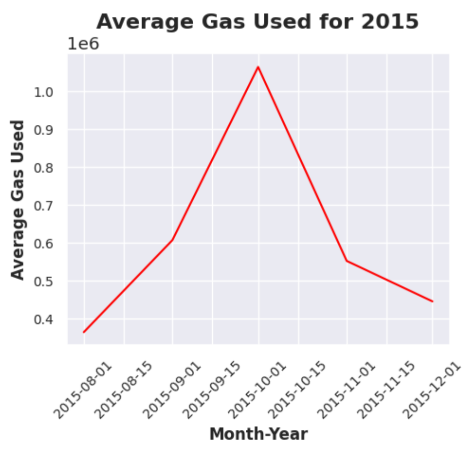
- The given average gas used data shows fluctuations over time, from 2015 to 2018.
- In 2015, the gas used values vary widely, starting with a significant spike in October, followed by gradually decreasing values. The year ends with a relatively lower value in December.
- In 2016, the average gas used experienced fluctuations, but within a narrower range compared to 2015. The highest gas used value occurred in May, while the lowest was seen in August. Throughout the year, the gas used values changed but without a clear upward or downward trend.
- In 2017, the average gas used values further fluctuated within a more compressed range compared to previous years. The highest value was observed in September, and the lowest was in January. Gas used values during the year show an almost linear pattern of increase.
- In 2018, the gas used values appeared to be more volatile than previous years, with a general upward trend from January to September. The highest value was recorded in April, while the lowest occurred around October. Toward the end of the year, the values seem to be following an increasing trend again.
- In conclusion, the average gas used on Ethereum has experienced fluctuations over the years. While there is no clear overall trend, there were periods of higher and lower gas usage throughout the given time frame.

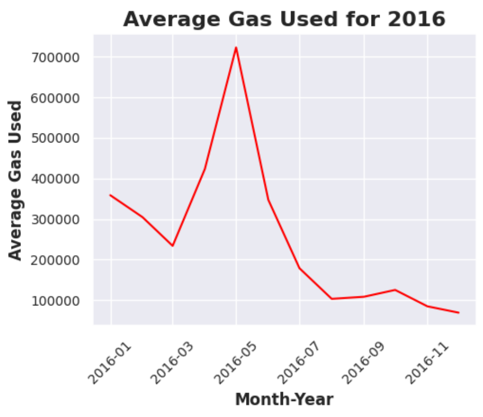


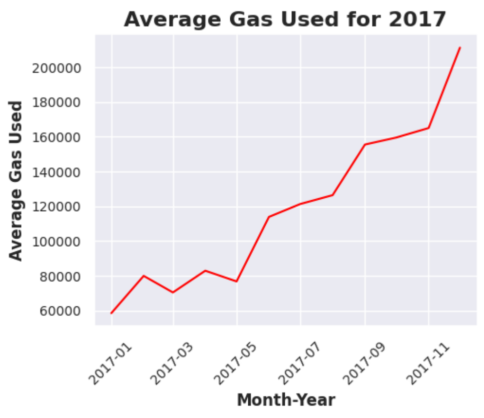


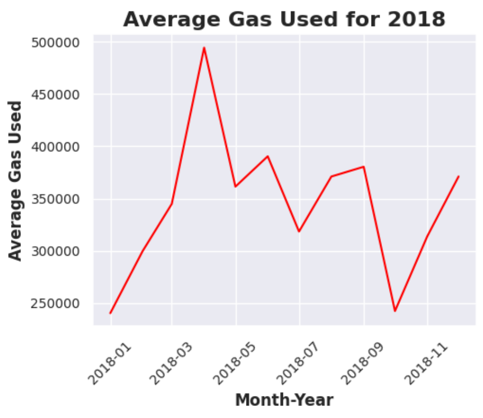












Gas use has generally increased over time, which may be a sign that contracts are getting more complicated and needing more gas to complete.

The data on gas consumption also indicates variations over time, which could be explained by a number of variables, including network congestion, changes in gas prices, and the adoption of more gas-efficient coding techniques.

Despite the general trend of rising gas consumption, there are occasional instances where it falls, which may be the consequence of advancements and advances in smart contract development.

The statistics on gas consumption shown here implies that contracts may be getting more complex generally and using more gas to execute them. The network situation, gas pricing, and advancements in contract development are a few other elements that also play a role in the observed variations in gas utilisation.

3. Miscellaneous Analysis - Data Overhead

Objective - The goal is to analyse how much space can be saved if logs_bloom, sha3_uncles, transactions_root, state_root, receipts_root columns were removed from the blocks table.

Relevant Source Code Files - overhead.py

Output Files - overhead_05-04-2023_16_03_33.txt and percentage_saved_05-04-2023_16_03_33.txt are the output file for overhead.py (same copy commands used to copy from bucket).

Execution Command - ccc create spark overhead.py -s -e 20

Relevant Event Log:

Version	App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
3.0.1	spark-1cac078bf8214111bcfd6b45f14ca46c	Ethereum	2023-04-05 17:01:30	2023-04-05 17:03:35	2.1 min	ec22387	2023-04-05 17:03:36

Explanation:

In overhead.py we accomplish the following.

- Read the blocks data from the S3 bucket into a DataFrame called blocks.
- Calculate the average length of each column in the blocks DataFrame (logs_bloom, sha3_uncles, transactions_root, state_root, and receipts_root).
- Compute the average size in bytes for each column, considering that each character after the first two requires four bits.
- Count the total number of rows in the blocks DataFrame.
- Calculate the total space saved by removing each column, by multiplying the average size in bytes for each column by the total number of rows.
- Create a DataFrame called results_df containing the total space saved for each column.
- Write the results_df DataFrame to a CSV file in the S3 bucket.

Total space saved is computed through the following method.

- For each column (logs_bloom, sha3_uncles, transactions_root, state_root, and receipts_root), subtract 2 from the average length to account for the '0x' prefix.
- Multiply the adjusted average length by 4 (bits per character) and divide by 8 (bits per byte) to obtain the average size in bytes.
- Multiply the average size in bytes for each column by the total number of rows in the dataset to compute the total space saved by removing that column.
- Store these results in a DataFrame called results_df with columns "column_name" and "size_bytes".

Here’s the output,

```
column_name        size_bytes
logs_bloom         1792000256
sha3_uncles        224000032
transactions_root  224000032
state_root         224000032
receipts_root      224000032
```

$$\begin{aligned}\textit{Total Space Saved} &= 1792000256.0\ (\text{logs\_bloom}) + 224000032.0\ (\text{sha3\_uncles}) \\ &+ 224000032.0\ (\text{transactions\_root}) + 224000032.0\ (\text{state\_root}) \\ &+ 224000032.0\ (\text{receipts\_root}) \\ &= 2688000384\ \text{Bytes} \\ &= 2.5034\ \text{Gigabytes (Approximately)}\end{aligned}$$

In the second part of the code,

- The code first calculates the average length of each column in the blocks dataset.
- It then calculates the average size in bytes of each column (considering each character after the first two requires four bits).
- The total number of rows in the dataset is calculated.
- The code then calculates the total space saved by removing each column by multiplying the number of bytes per row by the total number of rows in the dataset.
- The total space occupied by the dataset is calculated as the sum of space saved for all columns.
- The percentage of total space saved by removing each column is calculated by dividing the space saved for each column by the total space occupied and multiplying by 100.

The resulting output text file is read using read_csv() method of pandas. Then, we convert the space_saved column to float using astype(float) function. Then we divide this column by 1e6 to get values in MB (megabytes). This is done to get a nicer output.

	column_name	space_saved	percentage_saved
0	number	16.944447	0.465805
1	hash	224.000032	6.157792
2	parent_hash	224.000032	6.157792
3	nonce	56.000008	1.539448
4	sha3_uncles	224.000032	6.157792
5	logs_bloom	1792.000256	49.262338
6	transactions_root	224.000032	6.157792
7	state_root	224.000032	6.157792
8	receipts_root	224.000032	6.157792
9	miner	140.000020	3.848620
10	difficulty	45.092016	1.239586
11	total_difficulty	65.700399	1.806113
12	size	7.078047	0.194577
13	extra_data	114.808596	3.156104
14	gas_limit	17.414528	0.478728
15	gas_used	11.477951	0.315530
16	timestamp	28.000000	0.769724
17	transaction_count	-0.848591	-0.023328
18	base_fee_per_gas	0.000000	0.000000

Now we can get a better understanding of the impact of each column in terms of data overhead. It is readily apparent that logs_blooom column is the most impactful with nearly 50 percent of the data in blocks table coming from this column. Depending on the requirement of a particular application, all columns may not be necessary. For instance, the column extra_data contains arbitrary additional data, which might not be relevant to every analysis or application. Another example is nonce which is used for mining and Proof-of-Work, but if we are not analyzing mining patterns or the mining process itself, we can consider it less critical.

To summarize, the percentage_saved attribute provides a better view of the impact of each column in terms of data overhead.