ECS640U

BIG DATA PROCESSING

Coursework - Ethereum Analysis

Student number: 180510010

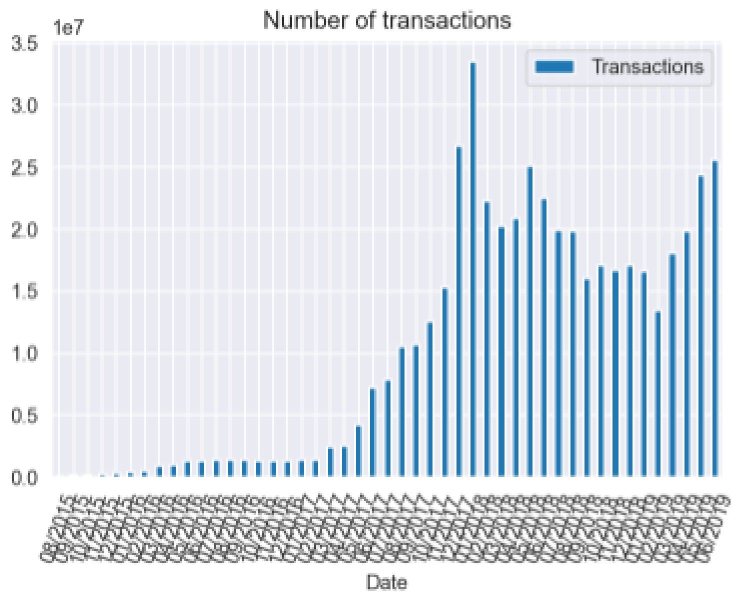
Note: Detailed explanation for steps taken in code is given as comments in code.
Code for all the graphs can be found in the Graphs.ipbyn file.

PART A Time Analysis

Graph values are sorted by year and month within the year.

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

partA1.py file has code for this section. Here we are using Hadoop map-reduce to perform this big data analysis.
This file consists of just one map-reduce job, taking input as the transactions table from /data/ethereum from our shared cluster space. We are counting the number of transactions for every month-year present in the data. The mapper splits the input line given, checks if it's a valid line ie if it has the correct number of fields as we are expecting it to have which will be 7 here for transactions data. We then get the timestamp fields and get the year-month combo in a single variable, from the valid lines. Yielding the year-month as key and 1 as value to the count the number of transactions. In the reducing stage, we are collecting the number of transactions based on year-month as the key and yielding the year-month as key and sum of counts ie the total number of transactions for that year-month as value.
We are also using a combiner to reduce the load on the reducer and the shuffle and sort stage, the code for which is the same as that of the reducer, adding the values and passing the key and new value pair forward.
The result of this is stored in partA1Output.txt file.
In the Graphs.ipbyn file we are loading the output file as a data frame (df1). That is being cleaned, pre-processed and then sorted based on year and month within the year group. The result is then being plotted as a bar graph as shown below.

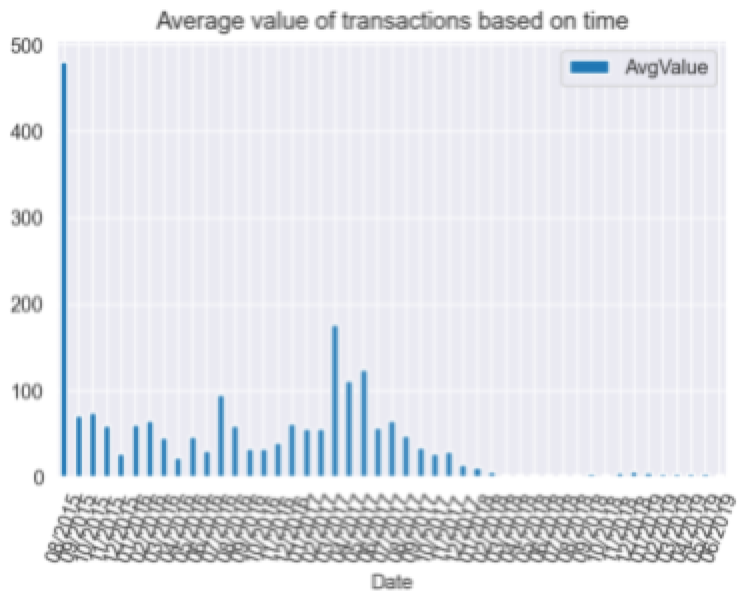


As we can see from the graph above the number of transactions increased steeply from 08/2015 to 01/2018 and then we can see the number of transactions going up and down. The peak point was January 2018. Then the graph shows a decrease in the total number of transactions when compared to peak but it was still better when compared to the earlier years.

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

The code for this part is in partA2.py file. We are giving the transactions table as an input here as well as we did in the previous step. Here we are calculating the average value of transactions taking place every month of the year using map-reduce.
This program contains one map-reduce job, mapper checks if it's a valid line or not and if yes then gets the timestamp and converts it to year-month variable ie we just get the month and the year from the timestamp converted date time, and get the value of transaction and yields year-month as key and a tuple of values as (value, 1). 1 as count as a way to keep a track of a number of transactions.
We are using a combiner to reduce the load onthe reducer, as we can't perform the average in the combiner cause we don't have all the data yet we will just accumulate data for the reducer. We will be yielding year_month as key, the sum of value yielded by mapper based on year-month key and total count of a number of transactions as a tuple of values.
In the reducer, in its first step similar to the combiner, we are aggregating all the values given from the combiner and then performing the average to find the average value of transactions based on the year-month key.
The output is stored in the partA2Output.txt file. In plots.ipbyn file we load this output as a data frame (df2), the df2 is being pre-processed and sorted according to month within

the year and years themselves. The results are then plotted as a bar graph shown below.



The graph shows the average value of transactions was peak during the start of this ie 08/2015 and then dropped drastically. It faced its ups and downs, decreased a lot in 2018s and 2019s.

PART B

Code for this part is mentioned in PartB.py file.
We are giving the input here as two table data ie /data/ethereum/transactions and /data/ethereum/contracts as we are going to need both to perform this task.
Here we are using MRStep to perform and coordinate our 2 map-reduce jobs. In the first job, we are joining the transactions with contracts to see if it's a smart contract or not (filtering based on smart contract or not) and aggregating value (wei) based on address.
In the mapper here we are first checking where the incoming line belongs to by checking the number of fields after splitting the line by comma. If it belongs to contracts we are getting the address field from there and yielding address as the key and a symbol to recognize it is from contracts like 'C' and 1 as count as values. If it belongs to the transactions table then we are getting the address, yielding it as key and 'T' as a symbol of this entry being from transactions table and getting the value of the transaction in wei and yielding these two as values.
In the reducer for job 1 we are checking, as both values from contracts and transactions having the same address will come to the same reducer, we need to check if that address belongs to both and collect all the values coming from transactions entry. Then yielding the address as key and sum of values as the value.

The second job is mainly about figuring out the top 10 here. The mapper here just carries forward the key and the value pair received from the previous job to the reducer as we can't have a reducer by itself we need to have a mapper to have a reducer, we are sending the key and value as a tuple of values as they need to reach the same reducer in order for this job to work for top 10 in all data. Also cause we need to be yielding (None, (key, value)) for it to be sorted in the reducer using the sorted function. In the reducer, we are sorting values based on value and yielding only the top 10 addresses and values.
The final output is given below…. (also can see it in the partBOutput.txt file)

Address	Total Ether received
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"	84155100809965865822726776
"0xfa52274dd61e1643d2205169732f29114bc240b3"	45787484483189352986478805
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"	45620624001350712557268573
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"	43170356092262468919298969
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"	27068921582019542499882877
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"	21104195138093660050000000
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"	15562398956802112254719409
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"	11983608729202893846818681
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"	11706457177940895521770404
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"	8379000751917755624057500

PART C

partC.py file has the code for this part of the assignment. We are using Hadoop map-reduce and MRStep to run multiple jobs in the same program.
We are giving the input as the blocks table from /data/ethereum.
The first job is to aggregate the miner and size. The mapper here first checks if the line is a valid line by splitting the fields and checking the number of fields. then gets the miner id and size and yields it as miner key and size value.
We are using a combiner, which is the same as the reducer that helps aggregate the size and yields the miner as key and total of size as value. The same for reducer aggregates the size and yields the miner as key and size as value.

In the second job, we are taking just the top 10 values of miners and their size for mining. The mapper here just passes the previous key and value as values and sends None as key so that they all go to the same reducer. We are using a combiner which is the same as a reducer where we will sort the values using the sorted function and in just the reducer we give out only the top ten of all the values.

Shown below is the output of the program that represents the miner id and size of their mining. (Also present in the file partCoutput.txt) (Output shown below is after trimming the null and double quotes)

Top 10 most active miners and their respective size of blocks mined:

```
                Miner address                         Size of blocks mined

0xea674fdde714fd979de3edf0f56aa9716b898ec8 -      23989401188.0
0x829bd824b016326a401d083b33d092293333a830 -  15010222714.0
0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c -   13978859941.0
0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5 -    10998145387.0
0xb2930b35844a230f00e51431acae96fe543a0347 -    7842595276.0
0x2a65aca4d5fc5b5c859090a6c34d164135398226 -    3628875680.0
0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01 -     1221833144.0
0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb -       1152472379.0
0x1e9939daaad6924ad004c2560e90804164900341 -  1080301927.0
0x61c808d82a3ac53231750dadc13c777b59310bd9 -   692942577.0
```

PART D

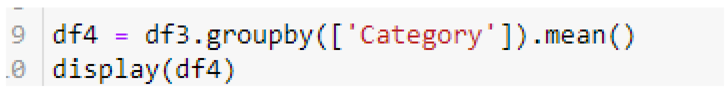
SCAMS:
Convert scams.json to scams.csv to make it easier for us to work with.
Using the scamJsonToCsv.py file we are converting the JSON file to a CSV file by parsing through the whole JSON file and adding its content to a new CSV file we will create. Can upload this file to the Hadoop cluster using -copyFromLocal and mention the path we want here, /user/ss389/scams.csv.

1. Popular Scams: Utilising the provided scam dataset, what is the most lucrative form of scam? How does this change throughout time, and does this correlate with certainly known scams going offline/inactive?

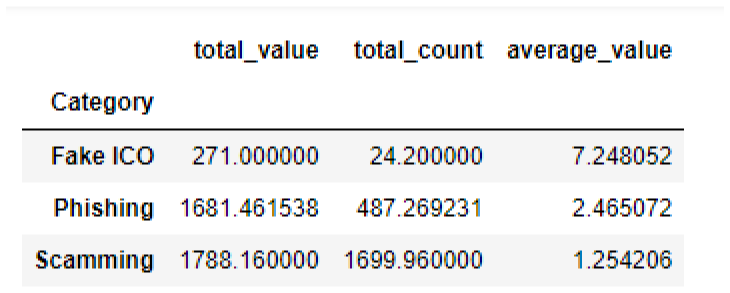
Here we are using spark. In ScamTransactionsAnalysis.py we first make a spark context. Then load our transactions table as a file and filter out bad lines ie those that don’t have fields we want. We do the same for the scams.csv file we just uploaded (ie check for bad lines).

Then we are taking the address and category from the scams loaded lines using a .map function.
With transactions good lines we are taking the to_address, value of the transaction, timestamp, and count. before returning from mapper we are converting the timestamp to get the year and month and then returning address as key and year_month, value and 1 (for a count of transactions to do the average later) as values.

Then we are joining the two sets of values we just extracted based on the address.
Then using a mapper we are setting our values in this order: category, year_month, value and count.
Using a reducer, we are adding up the value and count variables (the only numeric values here).
Finally, we are using a map to get the files in the form we want and rounding the large values like total value and performing average on total rounded up value.
Our output is stored in the file named: monthlyScamOutput.txt
the values are in the order of Category, year-month, total value(rounded up), total count, average value.
After loading the output file in a data frame (see Graphs.ipbyn file for loading, cleaning and other details regarding graphs) and using the group by the function we can see the following results.



```
df4 = df3.groupby(['Category']).mean()
display(df4)
```



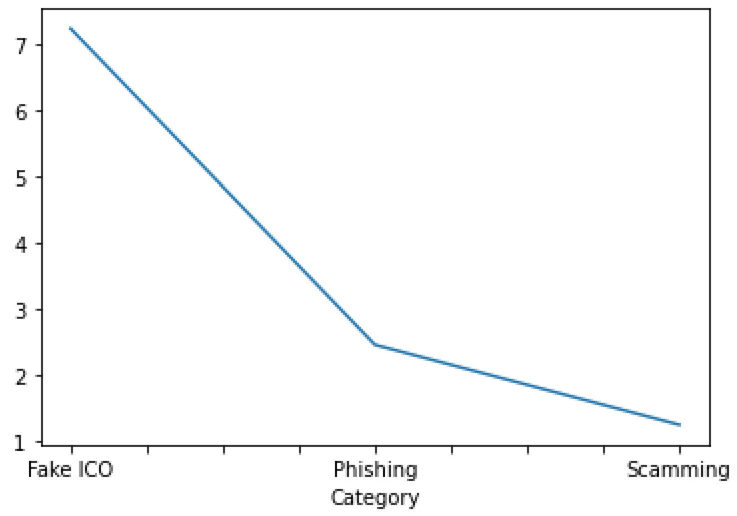
Category	total_value	total_count	average_value
Fake ICO	271.000000	24.200000	7.248052
Phishing	1681.461538	487.269231	2.465072
Scamming	1788.160000	1699.960000	1.254206

According to the data given, Fake ICO is/was the most lucrative form of scam. followed by Phishing and then Scamming.

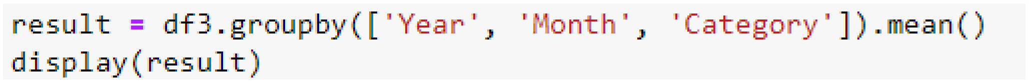
Based on the average value, we can also see it in the form of the graph below:

```python
df3.groupby(['Category'])['average_value'].mean().plot()
```

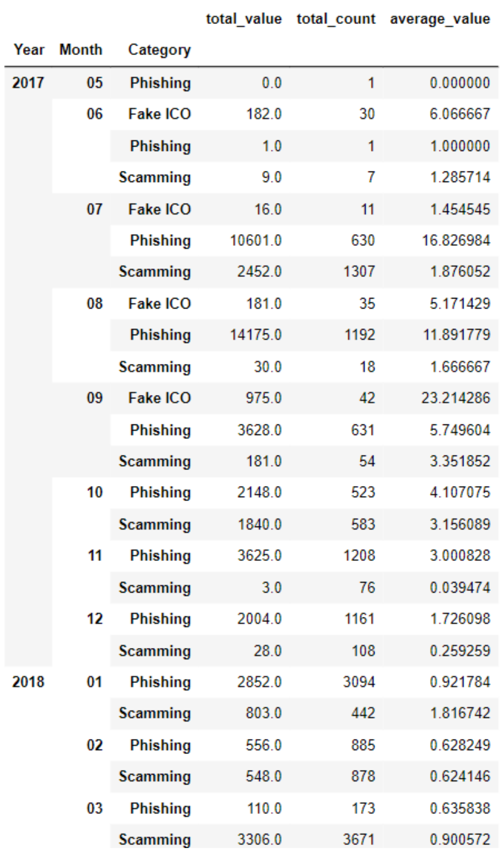
```
<matplotlib.axes._subplots.AxesSubplot at 0x23807d189a0>
```



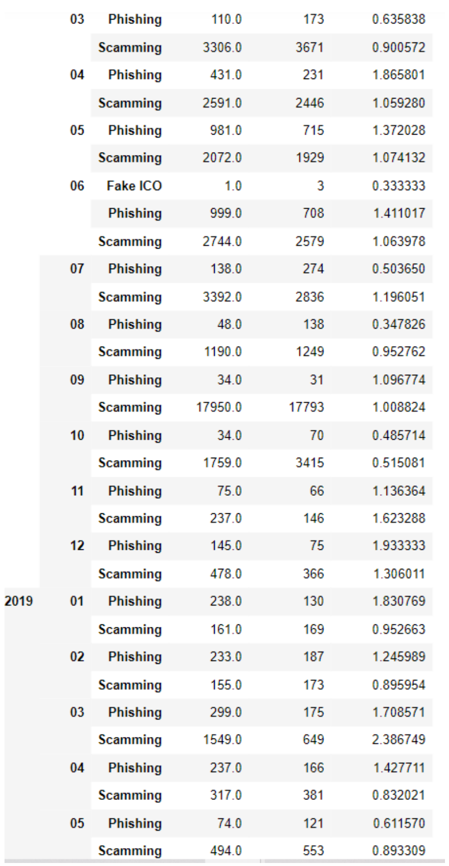
Analysis bases on time:



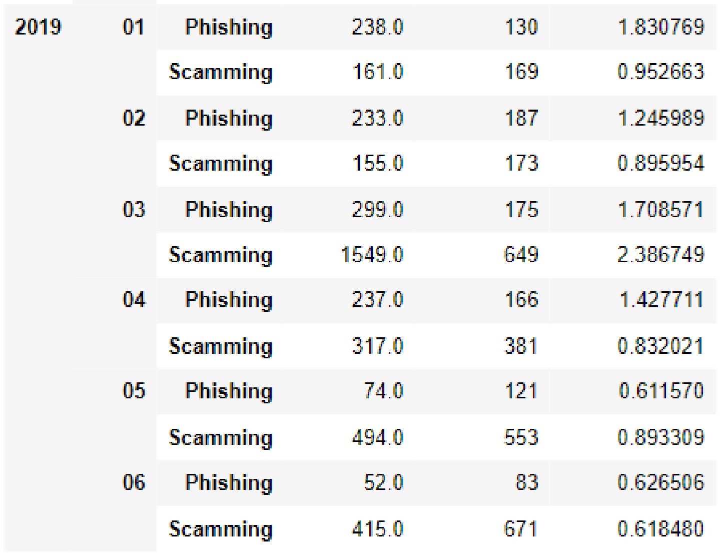
```
result = df3.groupby(['Year', 'Month', 'Category']).mean()
display(result)
```



Year	Month	Category	total_value	total_count	average_value
2017	05	Phishing	0.0	1	0.000000
	06	Fake ICO	182.0	30	6.066667
		Phishing	1.0	1	1.000000
		Scamming	9.0	7	1.285714
	07	Fake ICO	16.0	11	1.454545
		Phishing	10601.0	630	16.826984
		Scamming	2452.0	1307	1.876052
	08	Fake ICO	181.0	35	5.171429
		Phishing	14175.0	1192	11.891779
		Scamming	30.0	18	1.666667
	09	Fake ICO	975.0	42	23.214286
		Phishing	3628.0	631	5.749604
		Scamming	181.0	54	3.351852
	10	Phishing	2148.0	523	4.107075
		Scamming	1840.0	583	3.156089
	11	Phishing	3625.0	1208	3.000828
		Scamming	3.0	76	0.039474
	12	Phishing	2004.0	1161	1.726098
		Scamming	28.0	108	0.259259
2018	01	Phishing	2852.0	3094	0.921784
		Scamming	803.0	442	1.816742
	02	Phishing	556.0	885	0.628249
		Scamming	548.0	878	0.624146
	03	Phishing	110.0	173	0.635838
		Scamming	3306.0	3671	0.900572



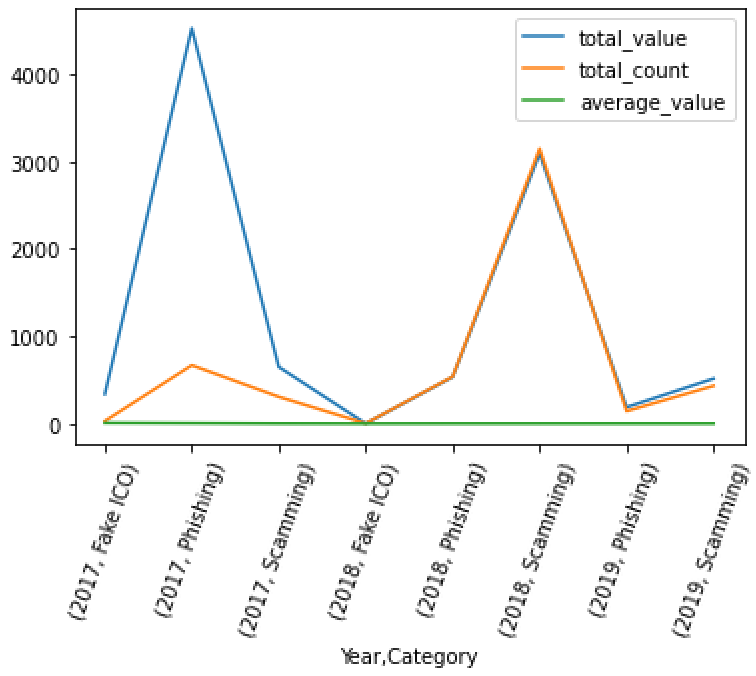
	03	Phishing	110.0	173	0.635838
		Scamming	3306.0	3671	0.900572
	04	Phishing	431.0	231	1.865801
		Scamming	2591.0	2446	1.059280
	05	Phishing	981.0	715	1.372028
		Scamming	2072.0	1929	1.074132
	06	Fake ICO	1.0	3	0.333333
		Phishing	999.0	708	1.411017
		Scamming	2744.0	2579	1.063978
	07	Phishing	138.0	274	0.503650
		Scamming	3392.0	2836	1.196051
	08	Phishing	48.0	138	0.347826
		Scamming	1190.0	1249	0.952762
	09	Phishing	34.0	31	1.096774
		Scamming	17950.0	17793	1.008824
	10	Phishing	34.0	70	0.485714
		Scamming	1759.0	3415	0.515081
	11	Phishing	75.0	66	1.136364
		Scamming	237.0	146	1.623288
	12	Phishing	145.0	75	1.933333
		Scamming	478.0	366	1.306011
2019	01	Phishing	238.0	130	1.830769
		Scamming	161.0	169	0.952663
	02	Phishing	233.0	187	1.245989
		Scamming	155.0	173	0.895954
	03	Phishing	299.0	175	1.708571
		Scamming	1549.0	649	2.386749
	04	Phishing	237.0	166	1.427711
		Scamming	317.0	381	0.832021
	05	Phishing	74.0	121	0.611570
		Scamming	494.0	553	0.893309



2019	01	Phishing	238.0	130	1.830769
		Scamming	161.0	169	0.952663
	02	Phishing	233.0	187	1.245989
		Scamming	155.0	173	0.895954
	03	Phishing	299.0	175	1.708571
		Scamming	1549.0	649	2.386749
	04	Phishing	237.0	166	1.427711
		Scamming	317.0	381	0.832021
	05	Phishing	74.0	121	0.611570
		Scamming	494.0	553	0.893309
	06	Phishing	52.0	83	0.626506
		Scamming	415.0	671	0.618480

```
df3.groupby(['Year', 'Category']).mean().plot(rot=70)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x23807d95ee0>
```



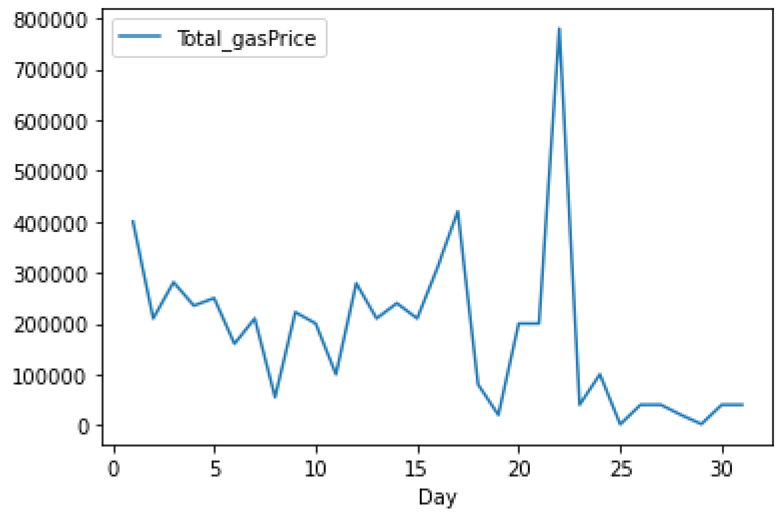
From the above graph, we can notice, over the period of time, the most popular scam, Fake ICO, became completely inactive in 2019 (nothing mentioned about it in the output data for the year 2019), while in 2018 Scamming become active and peaking. Phishing was peaking in 2017 but went low after, not as low as Fake ICO in 2018 though.

MISCELLANEOUS ANALYSIS:

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

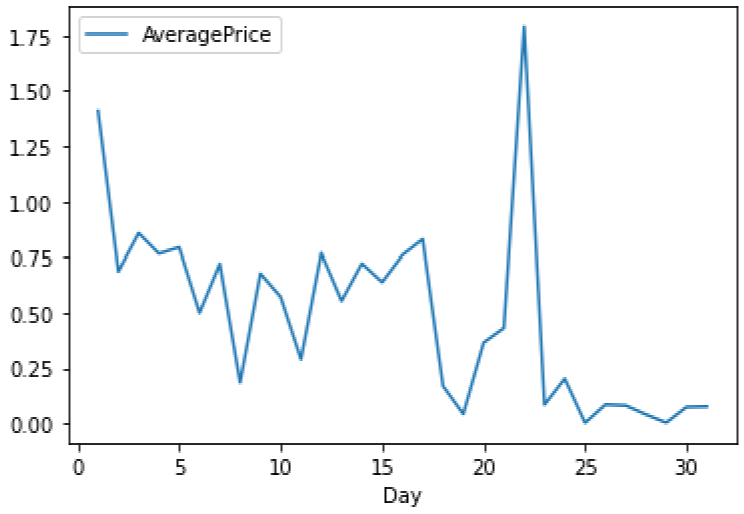
I selected the 2017 October (16th) fork and worked on it. The Byzantium fork. (details https://ethereum.org/en/history/#2017)
In file fork1.py we have one map-reduce job. Here we give the input as the transactions table and then in the mapper, we first check if the lines are valid or not after splitting the line based on commas and checking the size of the fields. Then we extract the date from timestamp and get just those having the year 2017 and month 10. We select those lines and yield the day ie the day of the month as key and gas price value and 1 for the count as a tuple of values.
We are using a combiner to reduce the load on the reducer. The code is the same as for the reducer. We are just summing up the count ie the 1s and the gas price values up based on the day of the month.
We are storing the output in fork1Output.txt file and loading it in Graphs.ipbyn file as a data frame and sorting it and cleaning up a bit. The below are the formed graphs and analysis results...

```
df6.plot(x ='Day', y='Total_gasPrice', kind = 'line')
plt.show()
```



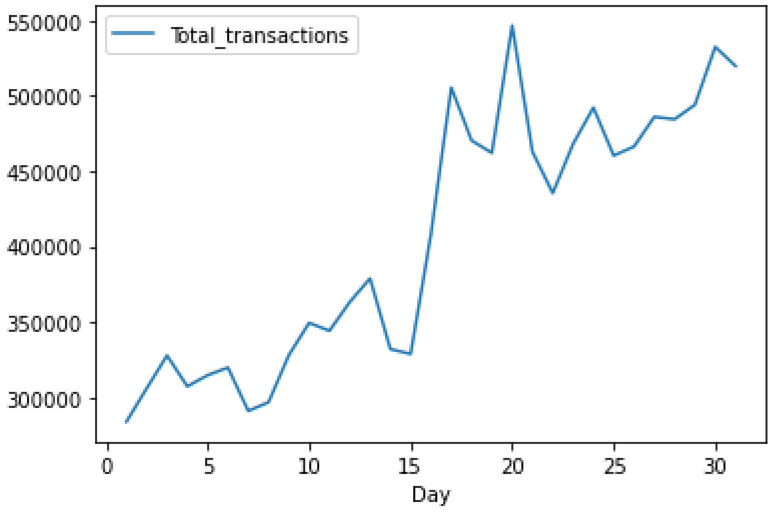
Average gas price value:

```python
df6['AveragePrice'] = df6['Total_gasPrice']/df6['Total_transactions']
df6.plot(x ='Day', y='AveragePrice', kind = 'line')
plt.show()
```



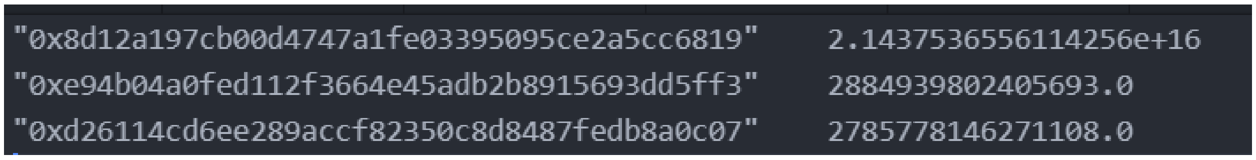
As we can see from the above graphs, the total gas price was at its peak on the 22nd of October 2017. Before the fork, as we can see the gas price was doing not bad but after the fork on the 16th the prices went up a bit and they dropped down and then peaked at the highest for that month. Gas prices faced low during the end of the month after the 25th.

```python
df6.plot(x ='Day', y='Total_transactions', kind = 'line')
plt.show()
```



The total number of transactions (general usage) also faced a great sudden increase on the 17th and then the highest peak for that month on the 20th. The total transactions did drop a little bit after the 20th but only to pick up and keep growing.

File fork2.py has Hadoop jobs to find who profited the most during this month. The first job is to collect all the to_addresses and values of transactions during this month and we are just summing it up in the combiner and reducer. In the second job, we are passing forward the key and value from our previous job and getting the top 3 addresses that have the highest values during this month. The result is stored in the fork2Output.txt file.



```
"0x8d12a197cb00d4747a1fe03395095ce2a5cc6819"    2.1437536556114256e+16
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"    2884939802405693.0
"0xd26114cd6ee289accf82350c8d8487fedb8a0c07"    2785778146271108.0
```

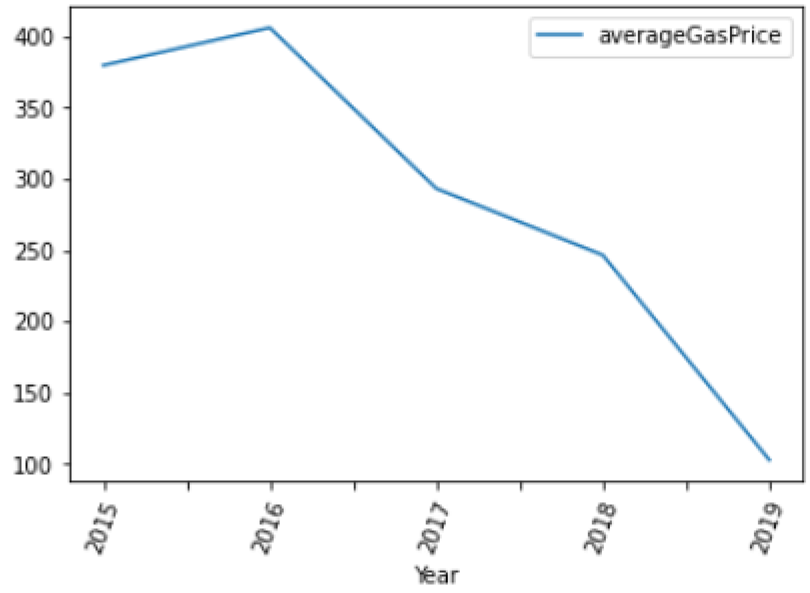
And as we can see the address "0x8d12a197cb00d4747a1fe03395095ce2a5cc6819" had received the highest value of wei, equal to 2.1437536556114256e+16. The other two are runner ups.

2. Gas Guzzlers: For any transaction on Ethereum a user must supply gas. How has the gas prices changed over time? Have contracts become more complicated, requiring more gas, or less so? How does this correlate with your results seen within Part B?

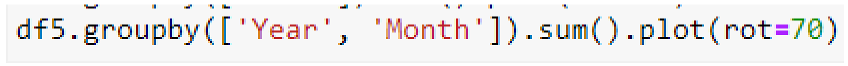
Gas price change over time:
Find code in GasAnalysis1.py. We are using Hadoop map-reduce for this. Here we are getting gas price per year_month combo.
In the mapper, we are first checking if this is a valid line and only then moving forward or else not considering that line. Then we are taking lines from the transactions table and getting the timestamp and convert it to get the year-month value. we are also extracting the gas price. The output of mapper is the year-month as key as we will be collecting average gas price based on year month and as values, we have the gas price and 1 as we will need the number of transactions to calculate the average at the reducer.

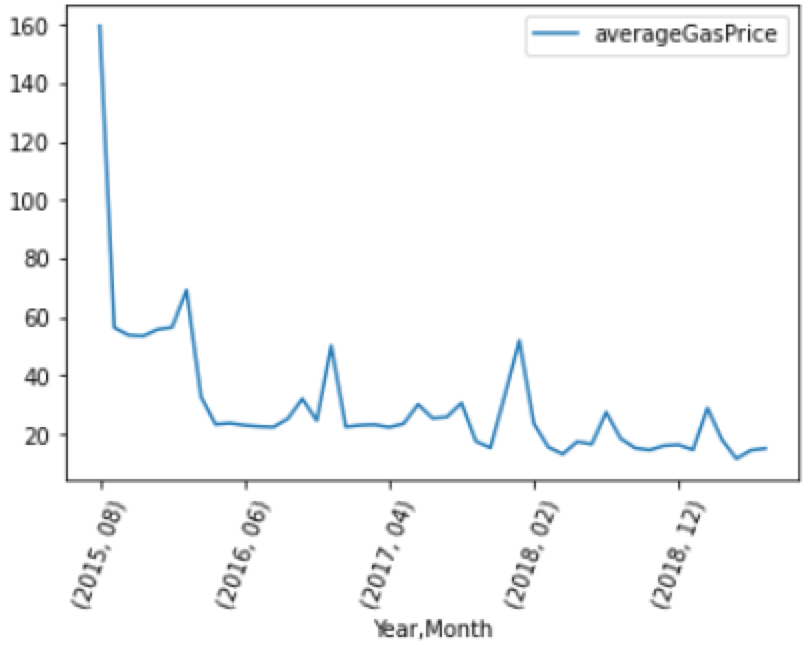
We are using a combiner to reduce the load on the reducer and shuffle and sort phase. As we need to perform an average task we can't do it in the combiner until we are aware of all the values coming from different mappers/ combiners. So we will just add up the count and gas price here, yielding year-month as key and total gas price and total count as values.
In the reducer, we are doing the total again but here at the end, we are also performing the average by dividing the total price by the total number of transactions. Hence finally yielding the year-month as key and average gas price as value.
We are getting the output file to your local machine by using -getmerge. Here find the output in GasAnalysis1Output.txt.
We then load the output file as a data frame and below are the analysis results…
[See Graphs.ipbyn file to see the cleaning and processing taking place.]

```
df5.groupby(['Year']).sum().plot(rot=70)
```



Above is a graph representing gas prices grouped by year.
Below is a graph representing gas prices grouped by year and month for a more detailed structured graph.


```
df5.groupby(['Year', 'Month']).sum().plot(rot=70)
```



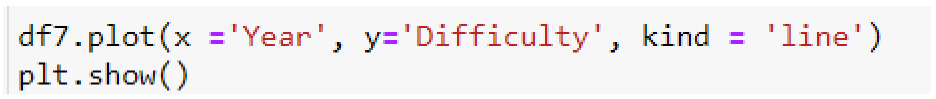
As we can see from both graphs, we notice a drop in gas prices over the period of time. Gas prices peaked in 2015 then faced a drastic drop in value after 2015 August, peaked again but not as much as before. Then the gas price gradually dropped while facing peaks at times but not much of a peak when compared to 2015 August.

GasAnalysis2.py has code for the second analysis regarding contracts and blocks.
We have 2 map-reduce programs here. In the first one, we are joining contracts and blocks together based on block number. In the second job we are going, to sum up the values based on timestamp converted to year-month.

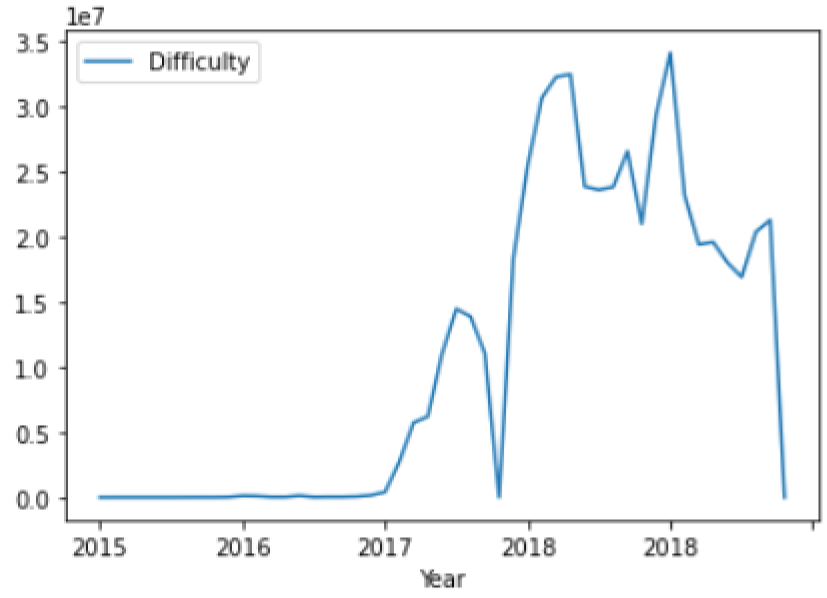
We are giving input as both the contracts and blocks data.
job 1:
In the mapper we are checking based on field length, after splitting with a comma, to see which table the line belongs to if the length is 9 then it belongs to the Blocks table, we are getting the block id, difficulty, timestamp converted to year_month and gas used from there and yielding block number as key and a symbol to represent; if it has 5 fields then it belongs to contracts table and we are getting the block_id from there and yielding the block id as key and a symbol to represent contracts and a 1 for the count, if not both then we discard the line.
In the reducer, as all same block id values come to the same reducer, we are checking if it exists in both blocks and contracts if yes then we are yielding year_month as key and difficulty and gas used as a tuple of values.

job 2 :
In mapper we are just passing on the values we gained from the previous job to the reducer.
In the reducer, we are just aggregating the difficulty and gas used. Yielding year_month as key and difficulty and gas used as values, which will be our final output.
Output can be found in GasAnalysis2Output.txt.
After loading the output in a data frame, and doing preprocessing like cleaning off the brackets and separating values, we can see the below results…

Year vs Difficulty:



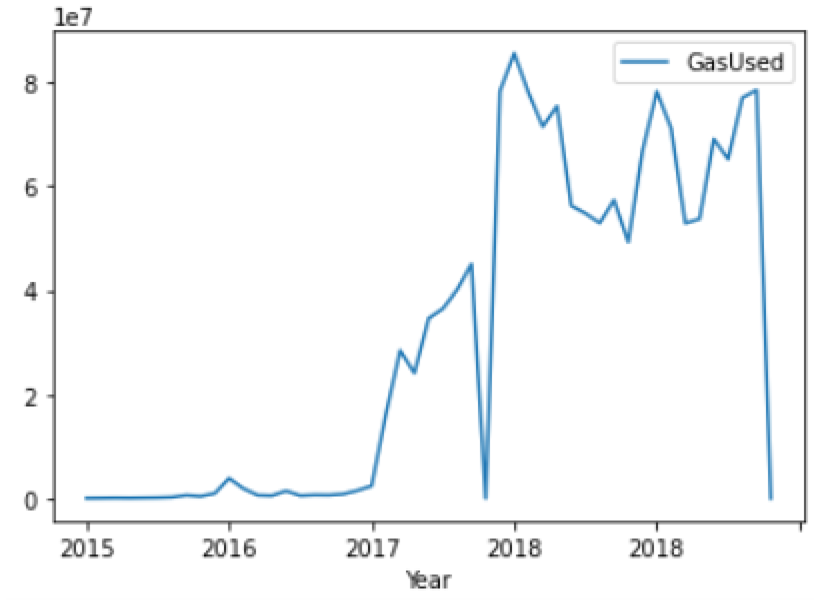
```
df7.plot(x ='Year', y='Difficulty', kind = 'line')
plt.show()
```



Difficulty of contracts increased over the years. It peaked at the end of 2018 and the start of 2019.

Year vs Gas used:

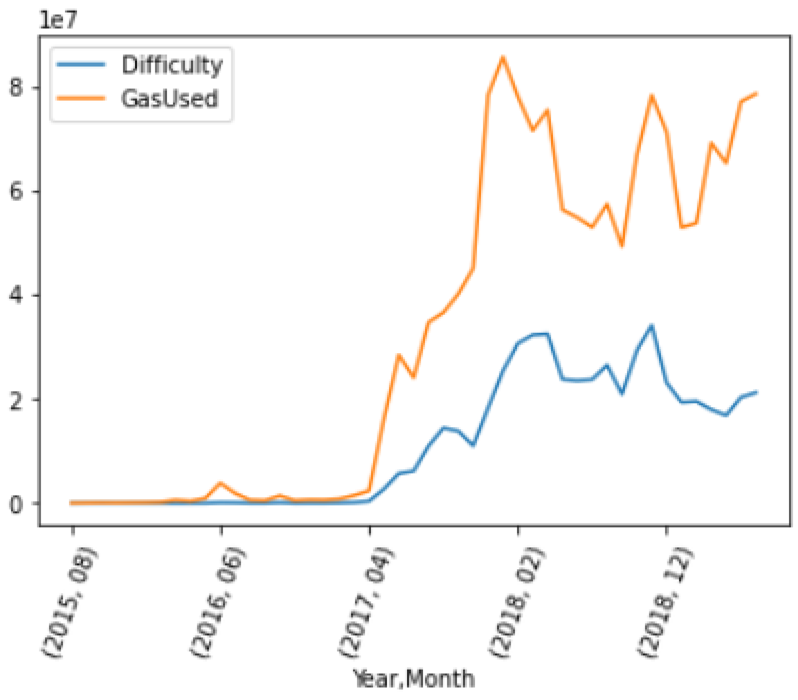
```
df7.plot(x ='Year', y='GasUsed', kind = 'line')
plt.show()
```



Usage of gas also increased over the period of years, but was at the highest during November of 2018 and though it faced a bit of up and down it was still around that point and didn't drop as low as during 2015 and 2016.

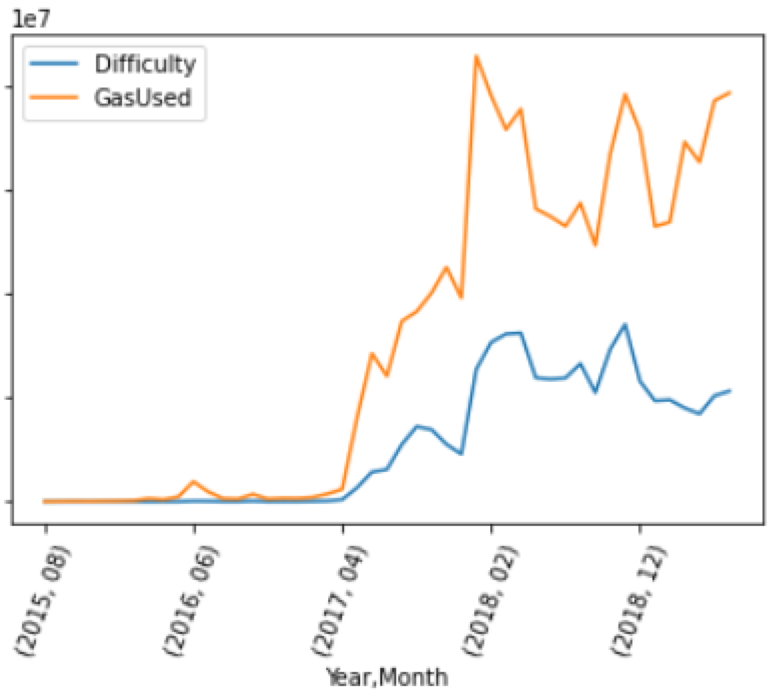
Sum of difficulty and Gas used Grouped by Year and Month:

```
df7.groupby(['Year', 'Month']).sum().plot(rot=70)
```



Average difficulty and Gas used grouped by Year and Month:

```
df7.groupby(['Year', 'Month']).mean().plot(rot=70)
```



From the above two graphs, we can see that both gas used and difficulty were very low from 2015 to April 2017 then both started rising.
Gas used had peaked in January 2018, it faced ups and downs but it never went back to its extreme low state again.
Difficulty peaked in December 2017 and it faced ups and downs but it peaked again in November 2018.
As we can see from both gas analyses parts 1 and 2, with the increase of gas using the gas price decreased, almost inversely proportionally.

3. Comparative Evaluation Reimplement Part B in Spark (if your original was MRJob or vice versa). How does it run in comparison? Keep in mind that to get representative results you will have to run the job multiple times, and report median/average results. Can you explain the reason for these results? What framework seems more appropriate for this task?

As we have done part B in Hadoop map-reduce here we are going to implement the same in Spark.
PartBSpark.py
In this code, as it is a spark program we need to have a spark context object made, then we are getting the lines from /data/ehtereum/transactions and /data/ethereum/contracts and checking if those are good lines or not line by line. Then we are getting what we need via the .map function. That is, we are getting the address and value from transactions and then reducing it to find the total value for that address.
As a result of that we are joining it with the contracts table (here we are selecting just the address from contracts and labelling it contract) based on address as a key and hence getting the values only for smart contracts as an output of this join. Then we use the takeOrdered function to get the top 10 values and print them out.
Here we can store the output in a file and compare it with our output from map-reduce by doing spark-submit partBSpark.py > partBSparkOutput.txt

We are also using a timer method to help us time the running of this program.
Time taken by spark programs is calculated via a timer function, that we can see in PartBSpark.py. There we are using the DateTime (module) library by python and get the difference in start time and end time ie before and after our spark method starts and after it ends (the output ie the difference is in the form of hours:minutes: seconds).

For Hadoop map-reduce we can check the execution time from the URL given in the terminal, for example:
  The URL to track the job: http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1637317090236_14879/
Through this we can go through our job histories and details and check the time (from the applications page), for each job and add time up for both the jobs in Hadoop for our part B, can also manually perform an average of time or use an online calculator.
I ran partB Hadoop again 5 times and made notes of the times.
Given below are the recorded times…
Time is given as HH:MM: SS (hours, minutes, seconds)

Job Type	Attempt 1 (time taken)	Attempt 2 (time taken)	Attempt 3 (time taken)	Attempt 4 (time taken)	Attempt 5 (time taken)	Average time taken	Median
Spark job	0:01:42	0:01:18	0:01:18	0:01:17	0:01:15	00:01:22	00:01:18
Hadoop job 1 + job 2	00:10:07	00:09:50	00:09:58	00:11:45	00:10:23	00:10:24	00:09:58

Here we can clearly see the **time taken** by Hadoop and spark for the same task. Spark took an average of 1 minute and 22 seconds whereas the Hadoop jobs took 10 minutes and 24 seconds.
The **main reason** is that in Hadoop we go and write the results to HDFS after job 1 and fetch it from HDFS for job2 which takes a lot of time. Whereas in Spark it's all based on RDDs that run in memory and don't need to fetch from far and we don't really come across a stage where we need to store intermediate results in a disk or other storage, at least not in this case.
As we can also see in the files containing the code, the **number of lines of code** for spark is less when compared to map-reduce jobs. It's just 8 lines of actual code (without counting the timer function and the import lines) while in Hadoop it is more or less 35 lines of code (without imports and main run part).
In spark, joining is so much **easier and less messy**, very straightforward and we just mention the key and other sets of values we are planning to join and it does it for us, we don't need to care about how it's implemented. Also to get the top10 we don't need a whole map-reduce job just for the top 10 when we can do it simply by using a function from Spark.

In almost all parameters, Spark is a more appropriate framework for this task.