Big Data Processing Project Report

In this report, I will present the required result and graphs of my findings for each task and explain in detail the logic behind each of the python file.

Please note that:

- The coursework zip folder contains python files, an excel file **data.xlsx** and another excel file **scams.csv** for Part D scams task and this report pdf file.
- the excel file **data.xlsx** contains all the output data and plots used for this coursework. The data and plots for each part are stored in a sub-sheet of this excel file with the name corresponding to the task name.

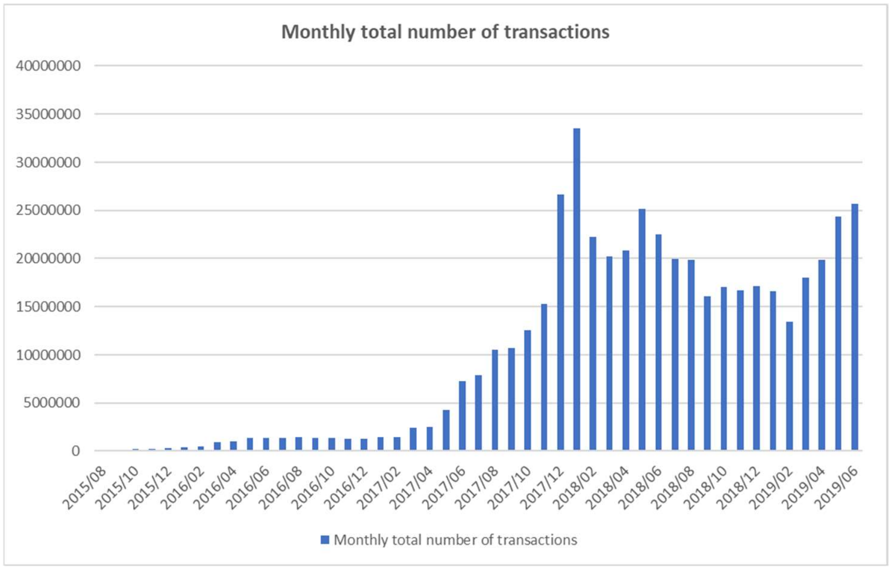
Part A - Time Analysis

In the first task of the part, I aim to generate the total number of transactions in each month over the whole period of the dataset. The python file to implement the task is '**parta_monthly_count_v2.py'**. The corresponding job ID is: **job_1637317090236_21545**. This python file contains a simple mapper-combiner-reducer combination. By looking through the **Transaction** dataset, I see that each line can be treated as a transaction, and the timestamp column can be used to get the unique year-month keys. Thus I can count the number of lines to represent the number of transactions.

In the mapper, the keys are the time in the format of 'year/month' which is based on the 7th column timestamp of the **Transaction** dataset with processing the data from timestamp format to readable time format by using the library 'datatime'. And simply emitting a value '1' to the combiner as a count indicator for counting the number of transactions. I believe by introducing a combiner it will ease the burden of massive data processing.

In the combiner, the keys to emit are still the year-month, and the values are the sum of all the '1' created previously for each key. Thus, at the end of this stage, it will count the total number of transactions for each unique keys in each combiner.

Lastly in the reducer, the logic is quite similar to the combiner with the keys as the year-month still, and the corresponding values are generated by calculating the final sum of transactions for each unique key – 'year/month'. As a result, it will be each unique year-month's total number of transactions. The result is shown in the graph below.

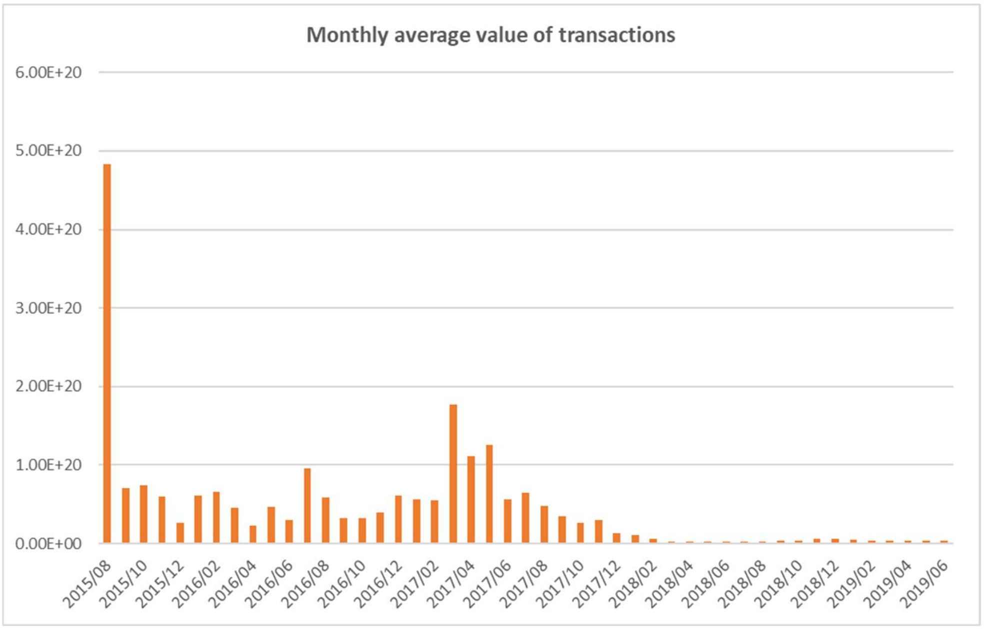


In the second half of Part A, I have created another python file '**parta_monthly_avg_v3.py**' to calculate every month's average value of transactions. The job ID is **job_1637317090236_21547**. Instead of calculating the number of lines of the transaction dataset, the monthly average value is calculated by using the aggregated value from the 4th column namely 'value' divide by the number of transactions.

In the mapper, the logic for the keys is the same as the previous python code as the 'year/month', and the values for passing to the combiner are in the format of (value, 1). The 'value' here is for later calculating the total value for each month, and '1' will later be used to calculate the number of transactions like the previous task.

Again, the combiner is being used to ease the burden of massive data processing. The year-month keys from the mappers will be passed to the combiner, and I will yield these keys to the reducer also. For the values to be yielded at this stage will be the sum of values and transaction numbers instead of performing the average operation. This is because the data at combiner is not the final global consolidated values for each unique key, the values are just the values in each combiner, it would be wrongly calculated if running the average operation here.

Lastly, in the reducer, keys are the unique year-month, and values are the corresponding calculated average values of transactions. The result is shown in the graph below.



Part B – Top Ten Most Popular Services

In this section, I am trying to find the top 10 aggregated value for each address which appears in both **Transaction** dataset and **Contract** dataset. The python file for implementing this task is '**partb_2.py**'. As join is needed between two large dataset, I decide to use Repartition join. The job IDs for this part are **job_1637317090236_21625** and **job_1637317090236_21634** for each of the step function.

In this python file, a two-step map-reduce combinations have been used. In the first step function, the goal is to output the keys of each address which appears in both **Transaction** and **Contract** dataset, and the corresponding values are the aggregated value from the **Transaction** dataset. In the second step, the goal is to output the top ten address-value pairs in the order of the value.

The first step contains a mapper and a reducer.

- The mapper emits two set of key value pairs from the **Transaction** dataset and **Contract** dataset separately. For the key/value pairs from the **Transaction** dataset, the keys are the 'to_address' field, and the values are from the 'value' field and a '1' is emitting with the value as an indicator to identify the dataset source of the key/value pairs are from the **Transaction** dataset rather than the **Contract** dataset. And for the key/value pairs from the **Contract** dataset, the keys are the 'address' field, and the values are (1,2), which only '2' is being meaningful for indicating the key/value pairs are from the **Contract** dataset.

- In the reducer, I have used 'if' condition functions with the previous '1' '2' indicators to identify if the key/value pairs are from which dataset. If it's from the **Transaction** dataset, I will aggregate the sum of values for the keys. And to use a Boolean variable 'smart' to set to 'True' when the address appear in the **Contract** dataset. And only when the value is >0 and 'smart' is True at the same time (which indicates that the key - address appears both in the **Transaction** and **Contract** dataset), the key/value pairs will be yielded then.

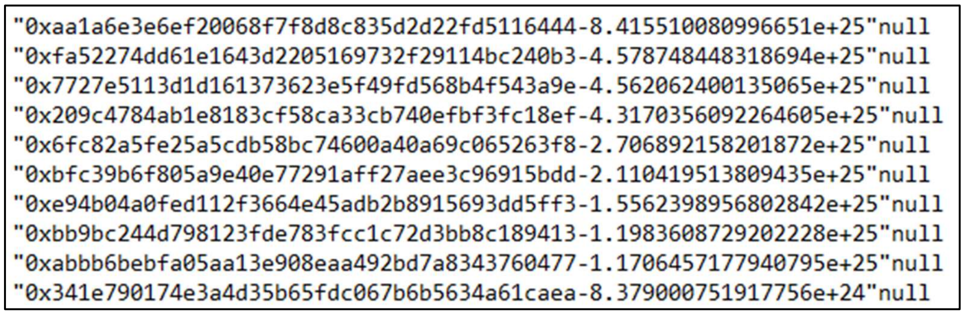
The second step contains a mapper, a combiner, and a reducer.

- In this mapper, the keys are just simple “top” text, and the values are in the format of (address, value) which are the key/value pairs from the previous step function.

- A combiner is used here to ease the burden of massive data processing. In this combiner, I have performed sorting to the values and only yield the top ten address/value pairs for each combiner.

- Lastly in the reducer, it is very similar to the combiner, to sort the values and yield the top ten address/value pairs both in the keys and leaving the value to ‘None’ in the end.

Please find the final output of the top 10 service as following:



```
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444-8.415510080996651e+25"null
"0xfa52274dd61e1643d2205169732f29114bc240b3-4.578748448318694e+25"null
"0x7727e5113d1d161373623e5f49fd568b4f543a9e-4.562062400135065e+25"null
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef-4.3170356092264605e+25"null
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8-2.706892158201872e+25"null
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd-2.110419513809435e+25"null
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3-1.5562398956802842e+25"null
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413-1.1983608729202228e+25"null
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477-1.1706457177940795e+25"null
"0x341e790174e3a4d35b65fdc067b6b5634a61caea-8.379000751917756e+24"null
```

Part C – Top Ten Most Active Miners

In this part, it is based on the **Block** dataset. The goal is to aggregate the ‘size’ field values for each unique ‘miner’, and then to find the top ten in size. Two-step map-reduce functions have been used for this task. The python file used here is ‘**partc.py**’. The corresponding job ID used is **job_1637317090236_21667** and **job_1637317090236_21668** for each of the step function.

The goal for the first step function is to output the ‘miner’ with its aggregated ‘size’. It contains a mapper, a combiner, and a reducer.

- In the mapper, it yields the key as the ‘miner’ field, and the value as the ‘size’ field.
- And for the purpose of ease the burden of big data processing, a combiner has been used. In the combiner, it simply calculates the sum of values for each key in the combiner. And then pass them to the reducer.
- In the reducer, it performs the sum of the value again for the ‘size’ for each unique key. And it yields the miner-aggregated size pair to the next step function.

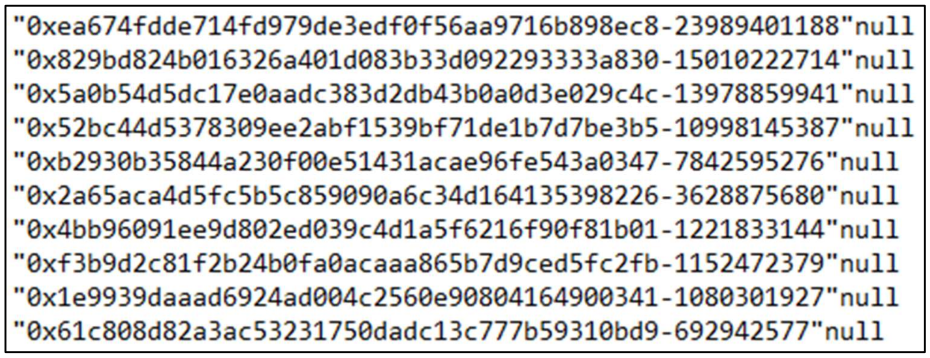
The goal for the second step function is to sort the top 10 miners. It also contains a mapper, a combiner, and a reducer.

- In the mapper, it yields “top” text as the keys, and it yields the value as the previous step’s key/value pairs in the format of (miner, aggregated size).

- And for the purpose of ease the burden of big data processing, a combiner has been introduced. In the combiner, it sorts the aggregated ‘size’ values for each combiner key which is the same text “top”, and keep the corresponding ‘miner’ values along with the ‘size’. And then only yields the top 10 values sorting by ‘size’. It in the end yields “top” text as the key, and it yields the top 10 (miner, aggregated size) for each key in the combiner.

- The reducer is very similar to the logic of combiner, but it calculated the top 10 ranking for each unique key globally. I have put the information all put in the output key and leave the output value as ‘None’. The output key is in the format that <miner> - <aggregated size>.

The final output is shown as following:

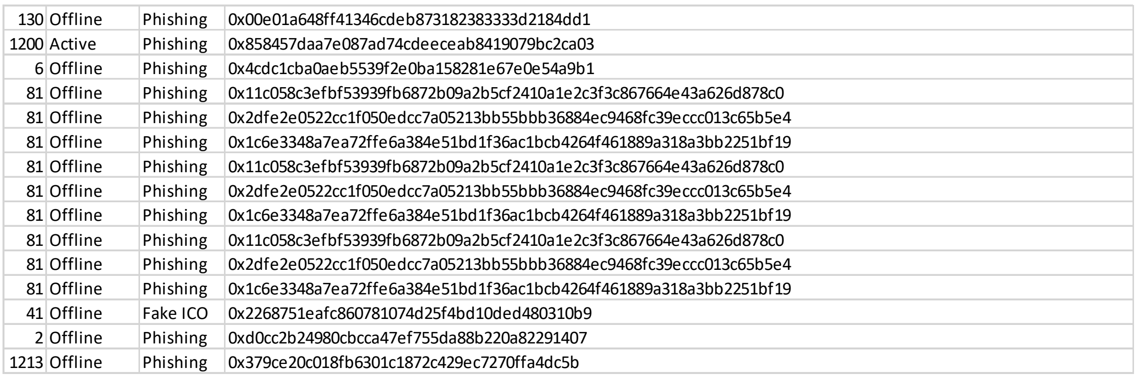


```
"0xea674fdde714fd979de3edf0f56aa9716b898ec8-23989401188"null
"0x829bd824b016326a401d083b33d092293333a830-15010222714"null
"0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c-13978859941"null
"0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5-10998145387"null
"0xb2930b35844a230f00e51431acae96fe543a0347-7842595276"null
"0x2a65aca4d5fc5b5c859090a6c34d164135398226-3628875680"null
"0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01-1221833144"null
"0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb-1152472379"null
"0x1e9939daaad6924ad004c2560e90804164900341-1080301927"null
"0x61c808d82a3ac53231750dadc13c777b59310bd9-692942577"null
```

Part D – Popular Scams

In this section, I am trying to explore what the most lucrative form of scam and if there is a correlation to the scam status.

Firstly, I have created a ‘**partd_json_2.py**’ python file to process the **scams.json** file and to convert it into a csv table. The table is called ‘**scams.csv**’. The table contains four columns. They are each scam’s ID, Status, Category and its address. A proportion of the table has been shown below.



130	Offline	Phishing	0x00e01a648ff41346cdeb873182383333d2184dd1
1200	Active	Phishing	0x858457daa7e087ad74cdeeceab8419079bc2ca03
6	Offline	Phishing	0x4cdc1cba0aeb5539f2e0ba158281e67e0e54a9b1
81	Offline	Phishing	0x11c058c3efbf53939fb6872b09a2b5cf2410a1e2c3f3c867664e43a626d878c0
81	Offline	Phishing	0x2dfe2e0522cc1f050edcc7a05213bb55bbb36884ec9468fc39eccc013c65b5e4
81	Offline	Phishing	0x1c6e3348a7ea72ffe6a384e51bd1f36ac1bcb4264f461889a318a3bb2251bf19
81	Offline	Phishing	0x11c058c3efbf53939fb6872b09a2b5cf2410a1e2c3f3c867664e43a626d878c0
81	Offline	Phishing	0x2dfe2e0522cc1f050edcc7a05213bb55bbb36884ec9468fc39eccc013c65b5e4
81	Offline	Phishing	0x1c6e3348a7ea72ffe6a384e51bd1f36ac1bcb4264f461889a318a3bb2251bf19
81	Offline	Phishing	0x11c058c3efbf53939fb6872b09a2b5cf2410a1e2c3f3c867664e43a626d878c0
81	Offline	Phishing	0x2dfe2e0522cc1f050edcc7a05213bb55bbb36884ec9468fc39eccc013c65b5e4
81	Offline	Phishing	0x1c6e3348a7ea72ffe6a384e51bd1f36ac1bcb4264f461889a318a3bb2251bf19
41	Offline	Fake ICO	0x2268751eafc860781074d25f4bd10ded480310b9
2	Offline	Phishing	0xd0cc2b24980cbcca47ef755da88b220a82291407
1213	Offline	Phishing	0x379ce20c018fb6301c1872c429ec7270ffa4dc5b

Secondly, I have created a two-step MRJOB python file ‘**partd_scams_v3.py**’ to produce several sets of output. One is scams’ category and its aggregated value based on the addresses in the scams, another one is the scam’s category-status and its aggregated value based on the addresses in the scams, and the last one is the scam’s status and its aggregated value based on the addresses in the scams. I need to do a join between the **scams.csv** and **transaction** dataset. I see that the csv file is a rather small file, so that it makes sense to apply a replication join.

In the first step function, it contains two mappers. The first one extract the data from the **scams.csv** file and form in a dictionary with the keys as the addresses, and the values as lists of three items – scam id, status and category. And in the second mapper, it goes through the **Transaction** dataset, and find the common addresses in both the **scams** and **Transaction** dataset and yield three sets of key/value pairs, which are category/value, status/value, and category-status/value.

In the second step function, it is just a simple mapper-reducer function. It calculated the aggregated value for these three sets of keys.

The job IDs are **job_1648683650522_3787** and **job_1648683650522_3789**. The raw outputs are shown as below.

Inactive1.488677770799503e+19
Offline4.002823599553411e+22
Phishing-Active4.5315978714979385e+21
Scamming3.840778126042357e+22
Scamming-Offline1.6292145589867755e+22
Scamming-Suspended3.71016795e+18
Suspended5.35007608e+18
Active2.664352337410128e+22
Fake ICO1.3564575668896302e+21
Fake ICO-Offline1.356457566889631e+21
Phishing2.692775739611062e+22
Phishing-Inactive1.488677770799503e+19
Phishing-Offline2.237963283877463e+22
Phishing-Suspended1.63990813e+18
Scamming-Active2.21119255026036e+22

I have then used excel to re-organise the data as below.

scam category	**aggreagated value**
Scamming	3.84E+22
Fake ICO	1.36E+21
Phishing	2.69E+22
scam category - status	**aggreagated value**
Fake ICO-Offline	1.36E+21
Phishing-Active	4.53E+21
Phishing-Inactive	1.49E+19
Phishing-Offline	2.24E+22
Phishing-Suspended	1.64E+18
Scamming-Active	2.21E+22
Scamming-Offline	1.63E+22
Scamming-Suspended	3.71E+18
status	**aggreagated value**
Offline	4.00E+22
Active	2.66E+22
Inactive	1.49E+19
Suspended	5.35E+18

From the first table, I see the aggregated value for each category of the scams. 'Scamming' has the highest volume. And by looking at the second table, it breaks down to each category's each status kind, it shows that 'Offline' scams contribute the highest volume for 'Phishing' scams whilst 'Active' scams contribute the highest volume for 'Scamming' kind of scams. In the last table for the summarised value for each status, 'offline' scams have the highest volume.

In conclusion, from these findings I believe status does have a correlation to the scams. The most lucrative form is in general 'Offline' status scams.

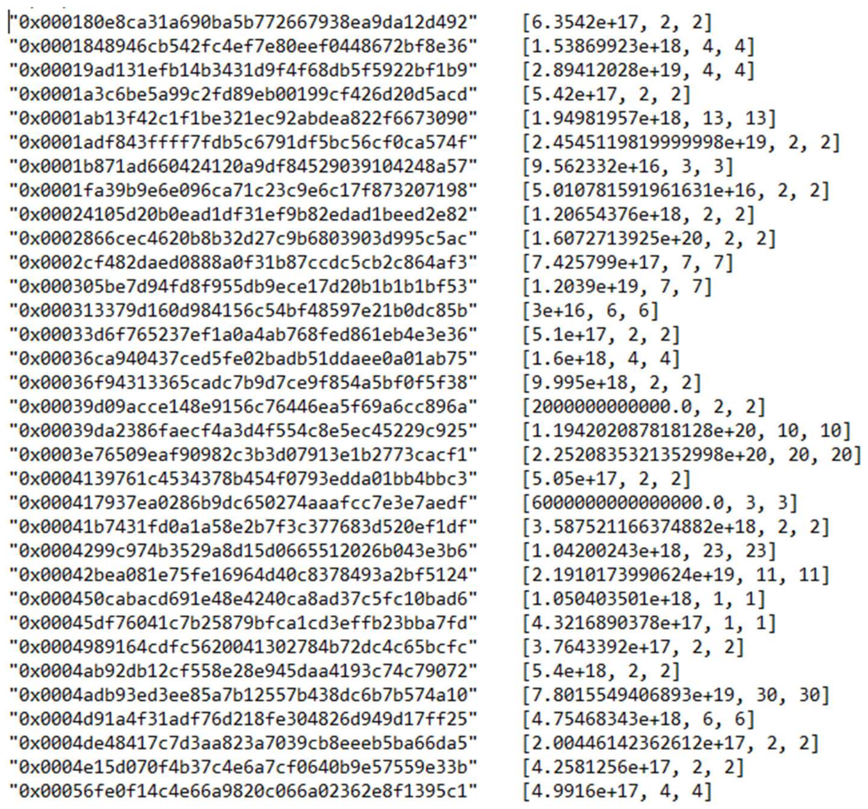
Part D – Contract Types (Partially done)

In this section, my goal is trying to output the aggregated value, number of transactions and number of outflow addresses for each 'to_address' which appears in both the **Contract** and **Transaction** Dataset. Since a join of two big dataset is involved, repartition join is more feasible to be applied. The python file to this task is called '**partd_contract.py**'. The job ID is **job_1648683650522_2665.**

This python file contains a simple mapper-reducer function. In the mapper, the number of fields of dataset has been used as a condition to identify if the data is from **Contract** or **Transaction** as their lengths are different. For the contract dataset, the key to yield to the reducer is the address, the value to yield is (1,1). Only the second 1 is useful, this '1' is used as an indicator to the dataset resource. And for the transaction dataset, the key to yield to the reducer is the 'to_address', the value to yield is((value, 1, from_address),2). The first part (value, 1, from_address) is used for later calculating the needed items as mentioned earlier, and the '2' is used as an indicator to the data resource.

In the reducer, I have used a Boolean variable 'smart' to set to True if the key/value pair is from the Contract dataset. And only when 'smart' is True and the aggregated value is positive, the values are yielded.

After I generated the output, I see that it is a massive data. The data volume is about 78MB. More big data processing maybe need to be applied in the future. Please find part of the data as shown below. The format of the output is (the received address, [aggregated value, number of transactions, number of sender address]).



```
"0x000180e8ca31a690ba5b772667938ea9da12d492"    [6.3542e+17, 2, 2]
"0x0001848946cb542fc4ef7e80eef0448672bf8e36"    [1.53869923e+18, 4, 4]
"0x00019ad131efb14b3431d9f4f68db5f5922bf1b9"    [2.89412028e+19, 4, 4]
"0x0001a3c6be5a99c2fd89eb00199cf426d20d5acd"    [5.42e+17, 2, 2]
"0x0001ab13f42c1f1be321ec92abdea822f6673090"    [1.94981957e+18, 13, 13]
"0x0001adf843ffff7fdb5c6791df5bc56cf0ca574f"    [2.4545119819999998e+19, 2, 2]
"0x0001b871ad660424120a9df84529039104248a57"    [9.562332e+16, 3, 3]
"0x0001fa39b9e6e096ca71c23c9e6c17f873207198"    [5.010781591961631e+16, 2, 2]
"0x00024105d20b0ead1df31ef9b82edad1beed2e82"    [1.20654376e+18, 2, 2]
"0x0002866cec4620b8b32d27c9b6803903d995c5ac"    [1.6072713925e+20, 2, 2]
"0x0002cf482daed0888a0f31b87ccdc5cb2c864af3"    [7.425799e+17, 7, 7]
"0x000305be7d94fd8f955db9ece17d20b1b1b1bf53"    [1.2039e+19, 7, 7]
"0x000313379d160d984156c54bf48597e21b0dc85b"    [3e+16, 6, 6]
"0x00033d6f765237ef1a0a4ab768fed861eb4e3e36"    [5.1e+17, 2, 2]
"0x00036ca940437ced5fe02badb51ddaee0a01ab75"    [1.6e+18, 4, 4]
"0x00036f94313365cadc7b9d7ce9f854a5bf0f5f38"    [9.995e+18, 2, 2]
"0x00039d09acce148e9156c76446ea5f69a6cc896a"    [2000000000000.0, 2, 2]
"0x00039da2386faecf4a3d4f554c8e5ec45229c925"    [1.194202087818128e+20, 10, 10]
"0x0003e76509eaf90982c3b3d07913e1b2773cacf1"    [2.2520835321352998e+20, 20, 20]
"0x0004139761c4534378b454f0793edda01bb4bbc3"    [5.05e+17, 2, 2]
"0x000417937ea0286b9dc650274aaafcc7e3e7aedf"    [6000000000000000.0, 3, 3]
"0x00041b7431fd0a1a58e2b7f3c377683d520ef1df"    [3.587521166374882e+18, 2, 2]
"0x0004299c974b3529a8d15d0665512026b043e3b6"    [1.04200243e+18, 23, 23]
"0x00042bea081e75fe16964d40c8378493a2bf5124"    [2.1910173990624e+19, 11, 11]
"0x000450cabacd691e48e4240ca8ad37c5fc10bad6"    [1.050403501e+18, 1, 1]
"0x00045df76041c7b25879bfca1cd3effb23bba7fd"    [4.3216890378e+17, 1, 1]
"0x0004989164cdfc5620041302784b72dc4c65bcfc"    [3.7643392e+17, 2, 2]
"0x0004ab92db12cf558e28e945daa4193c74c79072"    [5.4e+18, 2, 2]
"0x0004adb93ed3ee85a7b12557b438dc6b7b574a10"    [7.8015549406893e+19, 30, 30]
"0x0004d91a4f31adf76d218fe304826d949d17ff25"    [4.75468343e+18, 6, 6]
"0x0004de48417c7d3aa823a7039cb8eeeb5ba66da5"    [2.00446142362612e+17, 2, 2]
"0x0004e15d070f4b37c4e6a7cf0640b9e57559e33b"    [4.2581256e+17, 2, 2]
"0x00056fe0f14c4e66a9820c066a02362e8f1395c1"    [4.9916e+17, 4, 4]
```

Part D – Fork the Chain

In this section, my goal is to explore the impact from the DAO fork event which happened on 20/07/2016.

I have generated a python file ‘**partd_fork_1.py’** to output the daily aggregated value of transactions, daily total number of transactions and daily average gas price of transaction for the date 7 days before and after the event time. A simple mapper-combiner-reducer function has been used. I think a combiner can ease the burden of massive data processing. The Job ID is **job_1648683650522_2735.**

The raw output is as following:

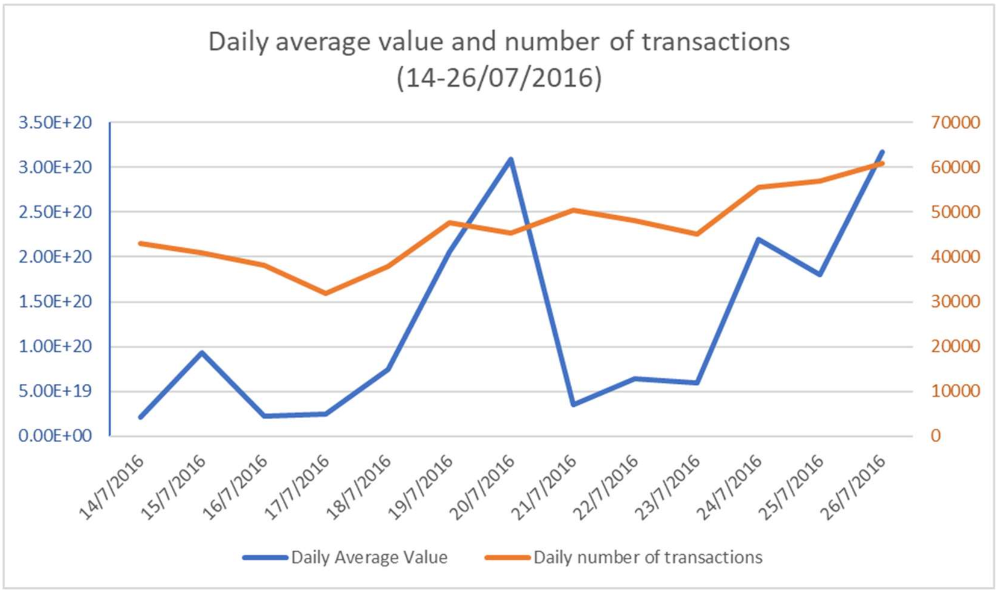
(in the format of: [date] [daily average value, daily count of transactions, daily average gas price])

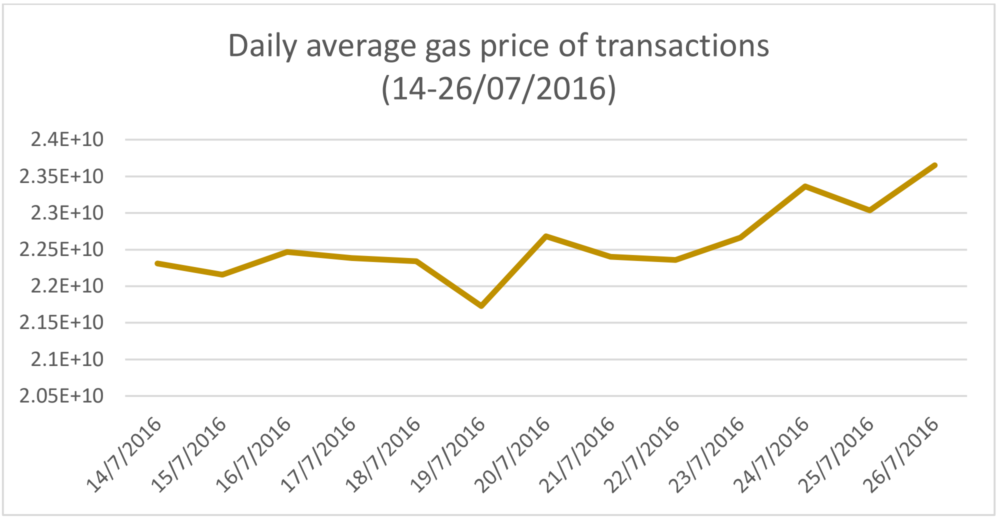
```
["14072016"][2.1638373796822426e+19, 42990, 22312198097.65278]
["16072016"][2.2199586801109225e+19, 38079, 22466795637.034374]
["18072016"][7.492362879610251e+19, 37813, 22338674827.95364]
["21072016"][3.5074500529992528e+19, 50544, 22401510908.645298]
["23072016"][5.930983247548485e+19, 45201, 22663572376.774807]
["25072016"][1.7994003736572667e+20, 56908, 23035732768.02583]
["15072016"][9.335170115292498e+19, 40915, 22156228061.678555]
["17072016"][2.455997956605008e+19, 31904, 22386531365.477932]
["19072016"][2.0521804980100566e+20, 47540, 21730371897.990112]
["20072016"][3.087404185748529e+20, 45234, 22680612752.35133]
["22072016"][6.381286395518237e+19, 48187, 22358106170.516487]
["24072016"][2.2010933094712643e+20, 55660, 23363782619.42551]
["26072016"][3.175146170165548e+20, 60995, 23653742896.17649]
```

I have then used excel to re-organise the data and produce the following two plots.

From the first plot, it shows a dramatic increase from 18 to 20 of July, and a dramatic drop on the following day. It matches the event time of DAO. And by looking at the trend for the daily number of transactions over the period, it increases steadily, and no suspicious fluctuation observed.

In the second plot, it shows the average gas price of transactions over the same period. It shows a sudden drop on the 19th and a sudden increase on the 20th, which matches with the event timeline.





Part D – Gas Guzzlers

In this section, I am trying to explore how has the supply of gas changed over time.

Firstly, similarly to Part B task, the only difference is that I have yielded the average gas of transactions along with other elements. I have generated the top 10 [address – value – average gas of transactions] in the order of value and for the common addresses appearing in the **Transaction** and **Contract** dataset. The python file to carry out this task is **partd_gas_1.py**. The Job IDs are **job_1648683650522_2762** and **job_1648683650522_2793** for each of the step function.

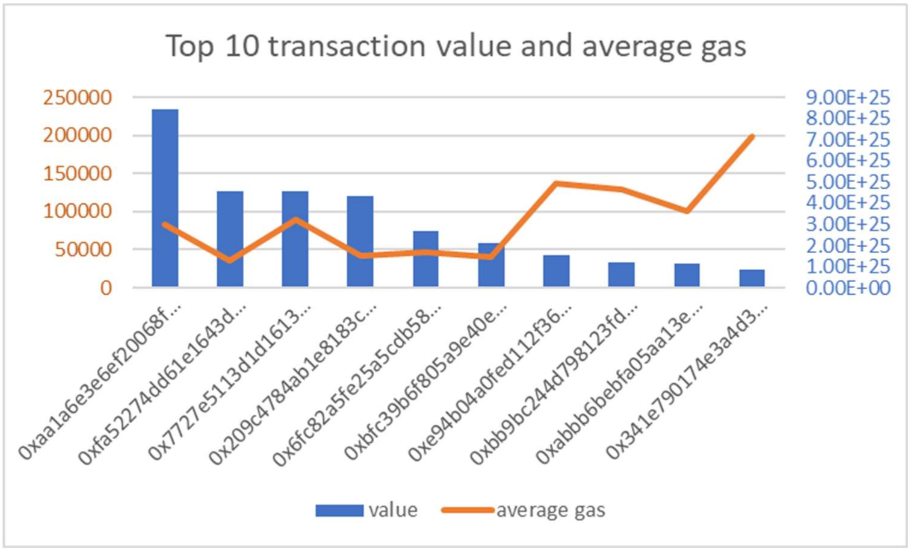
In the first step function, it contains a mapper-reducer to output keys as the address, and the corresponding values are (transaction value, gas, number of transactions). The number of transactions will be used to calculate the average gas in the next step function.

In the second step function, it contains a mapper-combiner-reducer. Again, by introducing a combiner can ease the burden of heavy big data processing. The final output presents in a way that all the information stores in the keys as the final top 10 addresses, aggregated value, and average gas of transactions. And set the reducer's value to 'None'.

The raw output is as shown below.

```
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444-8.41551008099601e+25-83155.65648193848null
0xfa52274dd61e1643d2205169732f29114bc240b3-4.578748448318875e+25-35000.037603453755null
0x7727e5113d1d161373623e5f49fd568b4f543a9e-4.5620624001352346e+25-89823.62314766804null
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef-4.317035609226825e+25-42761.8390420488null
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8-2.7068921582018533e+25-47007.97471560228null
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd-2.1104195138094966e+25-39999.61330880607null
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3-1.5562398956803897e+25-137076.8113569711null
0xbb9bc244d798123fde783fcc1c72d3bb8c189413-1.1983608729203506e+25-128910.86789658967null
0xabbb6bebfa05aa13e908eaa492bd7a8343760477-1.1706457177940896e+25-100337.97460384933nul
0x341e790174e3a4d35b65fdc067b6b5634a61caea-8.379000751917756e+24-198832.66666666666null
```

A plot has been produced for the top 10 values for a better read. It shows that the trend of average gas needed goes in the opposite direction comparing with the top 10 transaction values. I find it is not easy to draw some conclusions here.

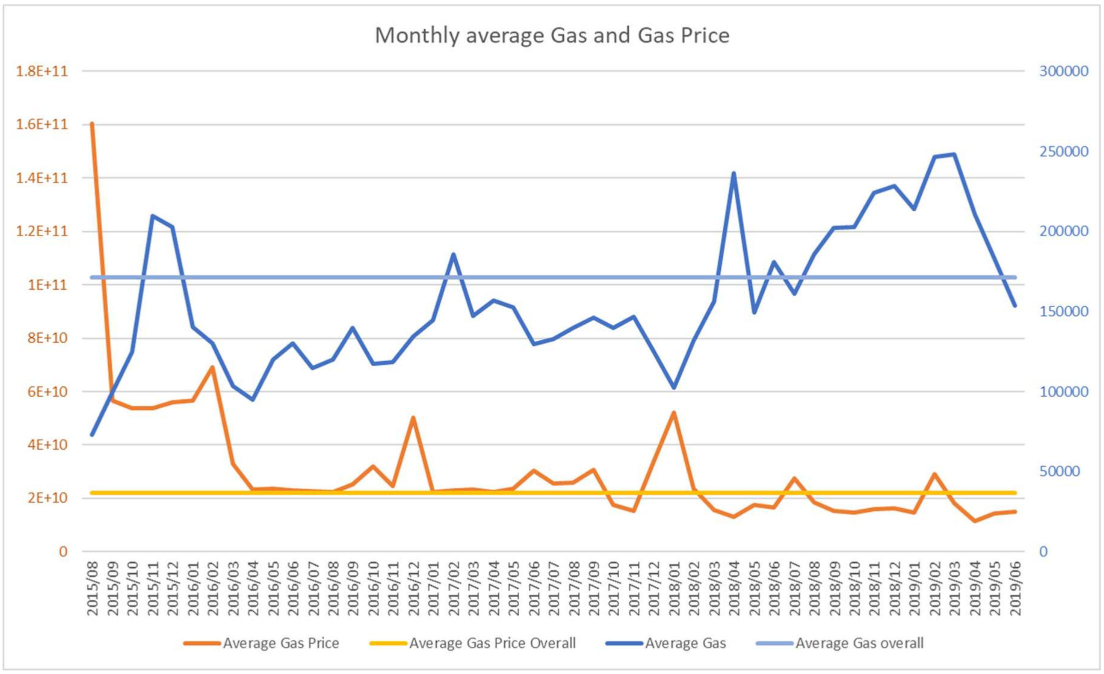


After that, I am thinking to output the monthly average gas and average gas price over the whole period to have another explore. The python file used for this task is '**partd_gas_2.py**'. The job ID is **job_1648683650522_2816**.

This python file contains a simple mapper-reducer combination. All the information is extracted from the **Transaction** dataset. There are four sets of outputs, they are the monthly average gas, the monthly average gas price, and the overall monthly average for the gas and gas price over the whole period.

I have processed the raw data and use excel to plot the below graph.

- For the gas price, I see that it started really high in 08/2015 and did not drop until 04/2016. And after that, it is quite stable comparing to the overall average gas price as approximately half of the data above the straight line and another half of the data below it.
- For the gas supply, it shows the monthly values quite match to the overall gas supply over the period.
- In conclusion, I believe that the gas supply is rather stable over the period.

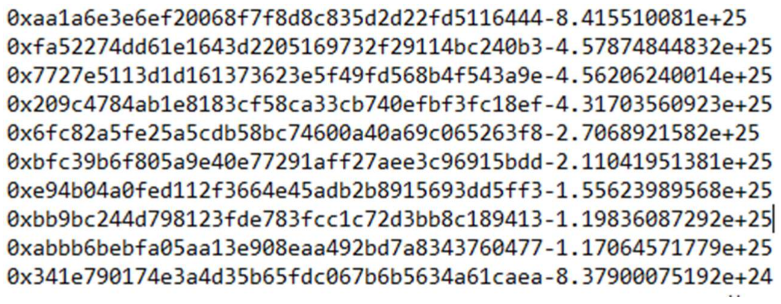


Part B by Spark

In this section, I will re-implement Part B's task by using Spark. The python file is called **partb_spark.py**. I have used this python file to run Spark RDD computations.

Firstly pyspark programmes need the 'sc' object to create the first RDD. After that, I have created two functions 'good_line_trans' and 'good_line_contracts' to define if the **Transaction** and **Contract** dataset is in the correct format. Then textFile() has been used to read from the required data path. After that filter() transformation has been used to run through each line in a way that only take into account the lines with True values from using predefined good line functions. After that map() has been used to perform a similar operation as mapper in MRJOB to producer key/value pairs, and followed by a reduceByKey() to combine all the values for each key (like a reducer function in MRJOB). After that, join() also has been used to join two tables by the same key. And then the top 10 value is computed by using the takeOrdered() function.

After running the code, I see the result output is the same as the result from Part B, which means my code works fine. The raw output is as following:



```
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444-8.415510081e+25
0xfa52274dd61e1643d2205169732f29114bc240b3-4.57874844832e+25
0x7727e5113d1d161373623e5f49fd568b4f543a9e-4.56206240014e+25
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef-4.31703560923e+25
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8-2.7068921582e+25
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd-2.11041951381e+25
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3-1.55623989568e+25
0xbb9bc244d798123fde783fcc1c72d3bb8c189413-1.19836087292e+25
0xabbb6bebfa05aa13e908eaa492bd7a8343760477-1.17064571779e+25
0x341e790174e3a4d35b65fdc067b6b5634a61caea-8.37900075192e+24
```

I have in total run three times for spark and mrjob codes to have a comparison on the time taken. The time has been recorded by using the linux 'date' command. The results and the corresponding Job IDs are shown in the below table.

	Time taken		Job ID	
	Spark	MRJOB	Spark	MRJOB
Test 1	1 min 49sec	25 min 59 sec	application_1648683650522_2932	job_1648683650522_2939, job_1648683650522_2945
Test 2	1 min 58 sec	25 min 40 sec	application_1648683650522_2933	job_1648683650522_2949, job_1648683650522_2951
Test 3	1 min 40 sec	27 min 04 sec	application_1648683650522_2935	job_1648683650522_2953, job_1648683650522_2969
Average Time	**1 min 49 sec**	**26 min 14 sec**		

From the test result, Spark only need averagely less than 2 mins to run the program whilst MRJOB needs averagely 26 minutes to run the program. Spark seems much more efficient for this task.

I think this is because for Spark many operations (transformations) are evaluated lazily such as map(), join(), reduceByKey() and etc. Transformations are only executed when they are needed. Only the invocation of an action is triggering the execution. It enables building the actual execution plan to optimise the data flow. And in addition, it is memory based which saves time when running it.

In contrast, the MRJOB python code involves two step function, it must run in a way the reducer needs to wait for the corresponding mapper's output in order to start the operation and running one step after the next step function. Also, data needs to be loaded from disk on every iteration and the results also need to save to HDFS with multiple replication which cost much more time than Spark. Therefore, Spark is more appropriate and more efficient to use than MRJOB for this task.