Ethereum Analysis

PART A – Time Analysis (part-a.py)

To complete this job, the transaction data set was loaded into an RDD called *lines*. The RDD was then filtered for malformed lines (lines where no. fields were not equal to 15). Next, the data in the RDD was mapped into nested tuples using Spark's map() function in the form

(month/year,(value, count = 1)

The reduceByKey() function was then used to sum the values and counts for a given month/year.

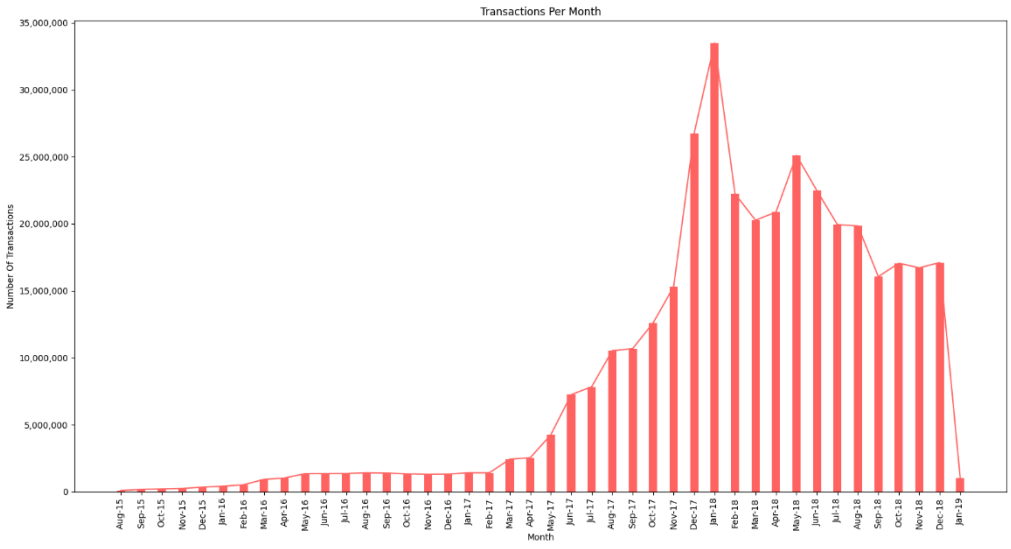
(*month/year*,(sum_values, sum_counts))

Finally, another map() was used to compute the average transaction value for each month producing tuples in the form of

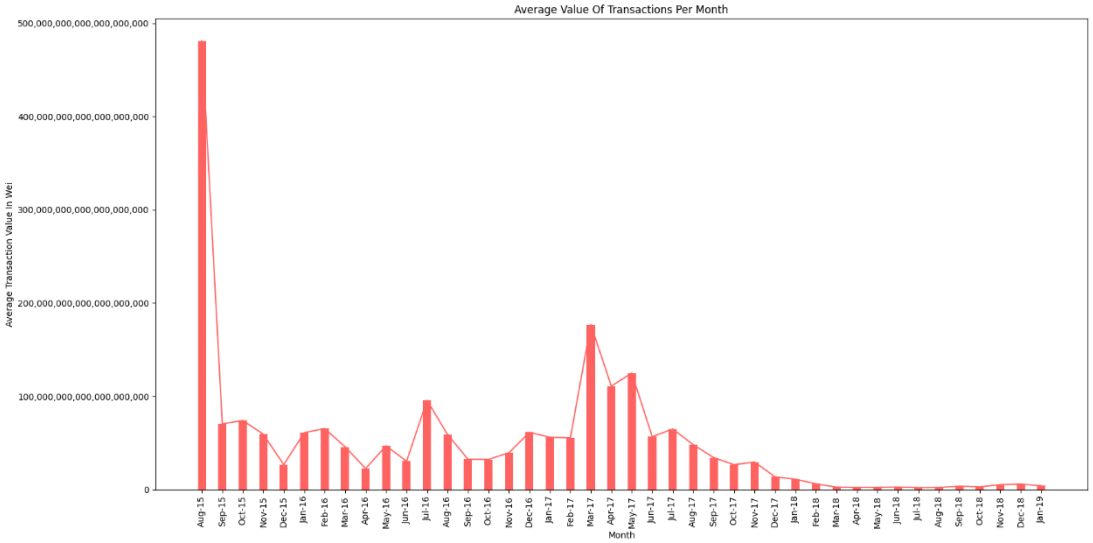
(month/year, avg_value, sum_counts)

Results

The number of transactions per month increased relatively slowly and appeared to stagnate below 2.5 million until February 2017. This increase was probably due to a surge in popularity of crypto currencies . Shortly after, in January 2018 a peak of just under 35 million transactions was recorded. After this, the number of transactions has tapered off slightly but remain relatively high into 2018 and beyond.



Initially, the average value of transactions was very high (nearly 500 Ether, 500×10^{18} Wei) perhaps due to the low conversion of Wei to normal currency like pound sterling but plummeted the month after to less than 100 Ether. Compared to the first two years of higher transactions values, 2018 saw the number of transactions per month flatline.



PART B - Top Ten Most Popular Services (part-b.py)

To compute the top 10 most popular services, both the transaction and contract data sets were required. Both were loaded into their respective RDD's and filtered for malformed lines similarly to the transactions data set in part A.

Relevant features were extracted into their own RDD's from each dataset. For transactions, the *to_address* field was mapped with the *value* to form a *(to_address, value)* pair. For contracts, the *address* field was mapped with *None* to for a *(address, None)* pairing.

The two RDD's were then joined to retrieve nested tuples in the form *(joined_address,(None, value))*

A new RDD was then created to hold *(joined_address, value)*. This RDD was then reduced to sum the value of transactions for a given address.

Finally, Spark's takeOrdered() was used to get an array of the top 10 addresses with the value of each tuple used as the sorting key.

Results

```
RANK:1  ADDRESS:0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444  VALUE:8415536369994176786737464 1
RANK:2  ADDRESS:0x7727e5113d1d161373623e5f49fd568b4f543a9e  VALUE:4562712851291534458774992 0
RANK:3  ADDRESS:0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef  VALUE:4255298913641319891929896 9
RANK:4  ADDRESS:0xfa52274dd61e1643d2205169732f29114bc240b3  VALUE:4054612845994729132622087 2
RANK:5  ADDRESS:0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8  VALUE:2454316173449977957116397 0
RANK:6  ADDRESS:0xbfc39b6f805a9e40e77291aff27aee3c96915bdd  VALUE:2110419513809366005000000 0
RANK:7  ADDRESS:0xe94b04a0fed112f3664e45adb2b8915693dd5ff3  VALUE:1554307763526374225471940 9
RANK:8  ADDRESS:0xbb9bc244d798123fde783fcc1c72d3bb8c189413  VALUE:1198360872910289384681868 1
RANK:9  ADDRESS:0xabbb6bebfa05aa13e908eaa492bd7a8343760477  VALUE:1071948594562894613652468 0
RANK:10 ADDRESS:0x341e790174e3a4d35b65fdc067b6b5634a61caea  VALUE:8379000751917755624057500
```

PART C - Top Ten Most Active Miners (part-c.py)

This required a simple map reduce job. The blocks data set was loaded into an RDD, filtered for malformed lines, mapped into *(miner, size)* pairs, used reduceByKey() to aggregate the sizes for a given miner and obtained top ten miners by using takeOrdered() using the *size* field as the sorting key.

Results

```
RANK:1  MINER:0xea674fdde714fd979de3edf0f56aa9716b898ec8 SIZE OF BLOCKS MINED:17453393724
RANK:2  MINER:0x829bd824b016326a401d083b33d092293333a830 SIZE OF BLOCKS MINED:12310472526
RANK:3  MINER:0x5a0b54d5dc17e0aadc383d2db43b0a0d3e029c4c SIZE OF BLOCKS MINED:8825710065
RANK:4  MINER:0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5 SIZE OF BLOCKS MINED:8451574409
RANK:5  MINER:0xb2930b35844a230f00e51431acae96fe543a0347 SIZE OF BLOCKS MINED:6614130661
RANK:6  MINER:0x2a65aca4d5fc5b5c859090a6c34d164135398226 SIZE OF BLOCKS MINED:3173096011
RANK:7  MINER:0xf3b9d2c81f2b24b0fa0acaaa865b7d9ced5fc2fb SIZE OF BLOCKS MINED:1152847020
RANK:8  MINER:0x4bb96091ee9d802ed039c4d1a5f6216f90f81b01 SIZE OF BLOCKS MINED:1134151226
RANK:9  MINER:0x1e9939daaad6924ad004c2560e90804164900341 SIZE OF BLOCKS MINED:1080436358
RANK:10 MINER:0x61c808d82a3ac53231750dadc13c777b59310bd9 SIZE OF BLOCKS MINED:692942577
```

PART D - Gas Guzzlers (part-d-gas.py)

Gas price vs time

For this section, I created a program which calculated the average gas price for each month.

The transaction data set was filtered and stored into an RDD. The map() function was used to create these three tuples. reduceByKey() was used to sum the gas prices and their counts for a given month/year.

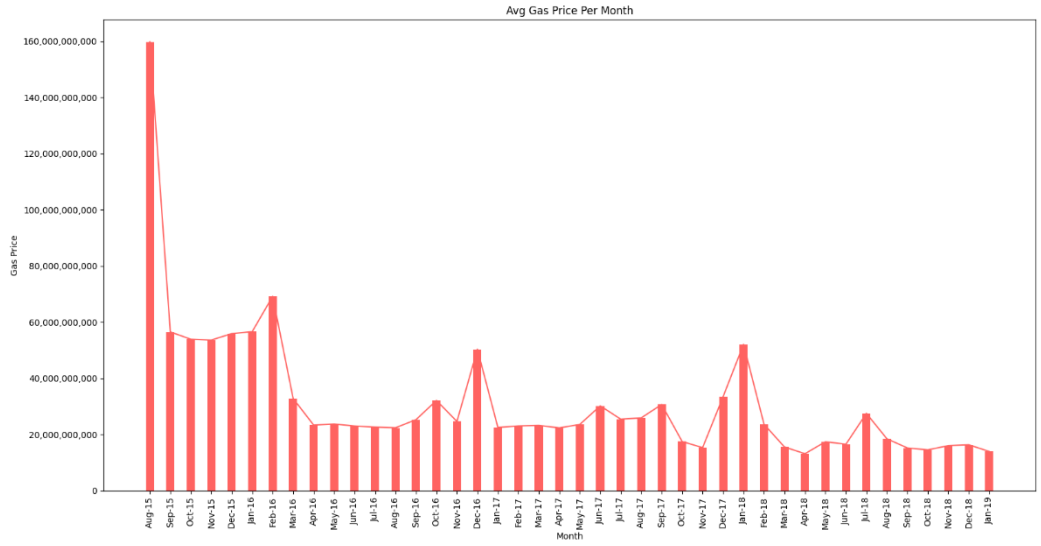
((month/year,(gas_price, count=1))

((month/year,(sum_gas_price, sum_counts))

((month/year, avg_gas_price)

Result

The average gas price drastically decreased from the start of the data set. However, there are noticeable peaks in the early and late months of each year.



Gas used vs time

A similar approach as with smart contracts in part b was taken. The contract and transaction data sets were joined, and the following features were extracted

(to_address,(gas, month/year)) -> From Transaction data set

(address, None) -> From Contract data set

(joined_address,(None,(gas, month/year))) -> Joined

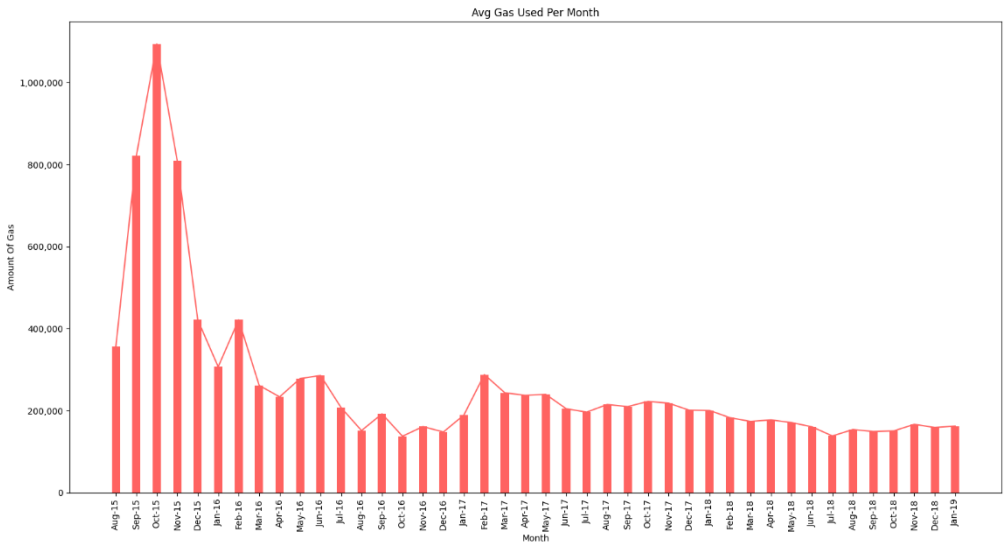
(month/year,(gas, count=1))

(month/year,(sum_gas, sum_counts))

(month/year, avg_gas)

Result

Initially, the average gas used per month was considerably but quickly tapered off and stabilised at around 200,000 units this would indicate that contracts have not become more complicated although the number of transactions has increased, perhaps the type of contracts being used are not gas guzzlers.



Top 10 popular contracts average gas used per month

To compute the average gas used by the top 10 smart contracts monthly, first the addresses of the top 10 contracts were stored in a set called "*addresses*".

The *to_address*, *block_timestamp* and *gas* fields were then extracted from transaction data set to give tuples in the form

(to_address,(month/year, gas))

The *address* field in contract data set was then mapped to None to produce tuples in the form

(address, None)

A join was then performed on the two rdd's giving nested tuples of

(joined_address,(None,(month/year, gas)))

From the joined rdd, the *month/year* and *gas* were extracted if the *joined_address* was present in the top 10 addresses set then a tuple of

(month/year,(gas, count=1))

Else

("other", (1, 1))

reduceByKey() is then used to sum the gas and counts for the months resulting in the rdd below

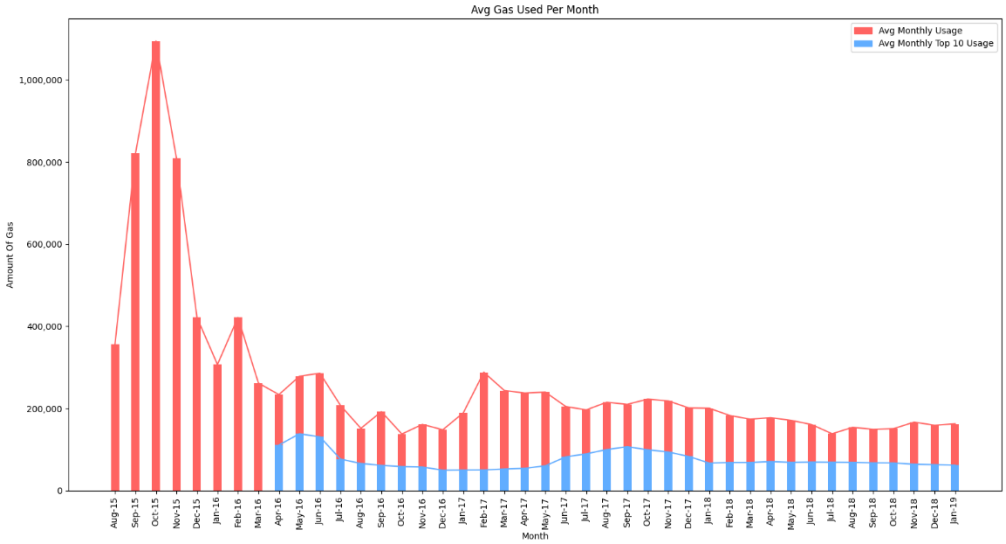
(month/year,(sum_gas, sum_counts))

A final map() was then used to calculate the average gas used per month for the top 10 contracts

(month/year, avg_gas)

.collect() was then used to get the average rdd into an array for printing of results.

Result



The average gas used for transactions per month came in at 264,293.0388 units and the average gas used per month across the top 10 smart contracts 74,846.58683 units.

For most months, the top 10 contracts alone accounted for 30-50% of the average gas usage indicating that these contracts use more gas than the average contract.

PART D - Data Overhead (part-d-overhead.py)

To calculate data overhead, 6 fields in the blocks data set were targeted: *logs_bloom*, *sha3_uncles*, *transactions_root*, *state_root*, *receipts_root* and *extra_data*.

The blocks data set was loaded into an RDD. Each block line was mapped using the calculate_bits() function. This function checks whether a field is relevant, takes the length of each hex string after the intial *0x* characters and mutliplies them by four with the assumption that each character after *0x* has a size of 4 bits and accumulates this value in the *useless_bits* variable. Finally a tuple of *('wasted_bits', useless_bits)* is returned.

reduceByKey() is then used to sum the *('wasted_bits', useless_bits)* tuples.

Result

My calculation gave me an output of 22422471840 bits ~ 2.8 GB saved if these 6 columns were removed from the blocks data set.

PART D - Contract Types (part-d-ctype.py)

For this section, contract types were distinguished between erc20 and erc721 from the contracts data set.

The *classify_contracts* function is used to map contracts into different categories depending on the values in the *is_erc20* and *is_erc721* fields. reduceByKey() was then used to sum the counts of pairs with the same label.

classify_contracts return tuples

is_erc20	*is_erc721*	Return
False	False	"NONE",1
True	False	"ERC20",1
False	True	"ERC721",1
True	True	"BOTH",1

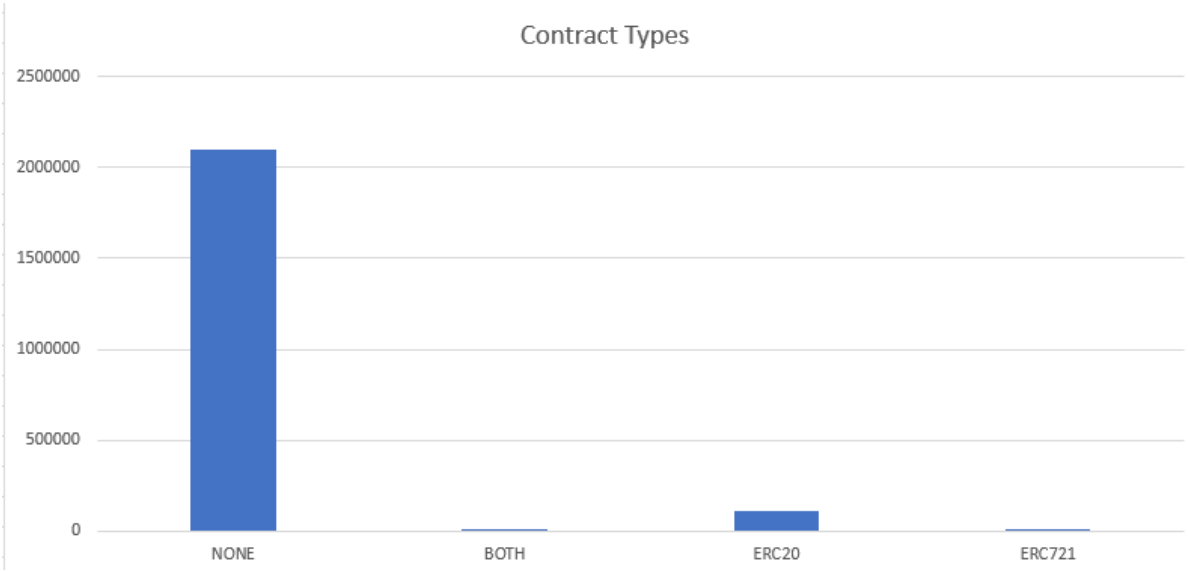
Result

NONE -> 2,103,509

BOTH -> 19

ERC20 -> 108,968

ERC721 -> 1913



There were significantly more erc20 contracts than erc721, perhaps non fungible tokens were not as popular yet.