Big Data Processing [ECS765P] - Ethereum Analysis

Jake Barrett - 190722595

SUBMITTED BEFORE THE INITIALLY PROPOSED DEADLINE (02/12/2019 at 10am)

PART A. TIME ANALYSIS (30%)

Here is a bar plot showing the number of Ethereum transactions occurring every month between the start and end of the *Transactions* dataset provided in HDFS:

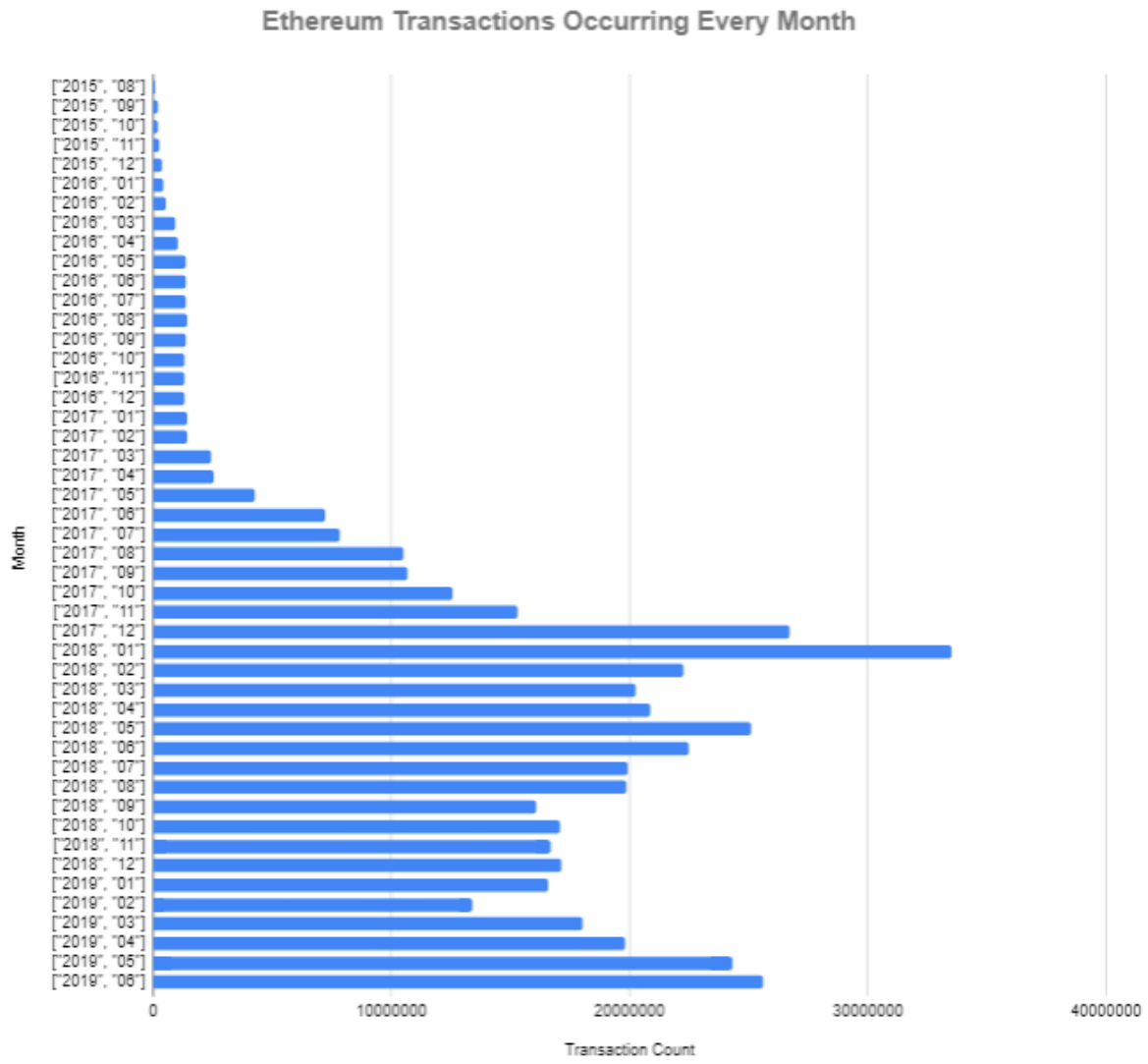

Figure 1: A bar plot showing the number of transactions occurring every month between the start and end of the dataset.

I use `MRJob` to execute the jobs in all parts of the Ethereum Analysis. Here are the MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if len(fields) == 7:
                time_epoch = int(fields[6])
                month = time.strftime("%m",time.gmtime(time_epoch))
                year = time.strftime("%Y",time.gmtime(time_epoch))
                key = (year, month)
                yield(key, 1)
        except:
            pass
```

We take the `/data/ethereum/transactions` dataset as our input and split the lines into fields where they are being comma-separated for those inputs which have precisely 7 fields since this is how many fields the *Transactions* dataset has. We want the sum of the transactions over each month, so we need to change the values of the `block_timestamp` field, which is in epoch time, to months by using the `strftime` operation. I stored the year and the month in a single key and yielded this key with the count value 1. We pass all the irrelevant information, such as the headings of the fields which would produce errors when executing this job.

Reducer

```
    def reducer(self, key, trans_count):
        yield(key, sum(trans_count))
```

We now take the output of the mapper as the input of the reducer and we simply yield the key being each distinct month with the summed transaction counts so we output the transactions per month.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574171293853_1496/

PART B. TOP TEN MOST POPULAR SERVICES (40%)

JOB 1 - INITIAL AGGREGATION

Firstly, we aggregate the *Transactions* to see how much each email address within the userspace has been involved. We do this by taking the aggregate on the field `value` for each `to_address`.

Here are the MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if len(fields) == 7:
                to_addr = fields[2]
                value = int(fields[3])
                yield(to_addr, value)
        except:
            pass
```

We take the `/data/ethereum/transactions` dataset as our mapper input and as in *Part A*, we split the lines into fields where they are being comma-separated and considering those inputs with field length 7. We filter the data so that we yield the fields `to_address` and `value` and we pass all irrelevant information.

Reducer

```
    def reducer(self, to_addr, value):
        yield (to_addr, sum(value))
```

Then the reducer takes the output of the mapper as its input and we yield the `to_address` along with the sum of the values in `value` for each `to_address`.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574171293853_2137/

JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING

Now we have obtained the *Transaction* aggregate, the next step is to perform a repartition join between this obtained aggregate and *Contracts*. We'll do this by performing a join on the `to_address` field of the output of *Job 1* with the `address` field of *Contracts*.

Here are the MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            #one mapper, we need to first differentiate among both types
            if len(line.split(','))==5:
                # line = line.strip()
                fields = line.split(',')
```

```
                join_key = fields[0]
                join_value = int(fields[3])
                # print(join_key)
                yield (join_key, (join_value,1)) # 1 for CONTRACTS


            elif len(line.split('\t'))==2:
                # line = line.strip()
                fields = line.split('\t')
                join_key = fields[0]
                join_value = int(fields[1])
                join_key = join_key.replace('"', '')
                yield (join_key,(join_value,2)) # 2 for JOB1 OUTPUT
        except:
            pass
```

We will have 2 inputs in our mapper being the output from *Job 1* and the `/data/ethereum/contracts` dataset. We essentially create two mappers in one by introducing `if` and `elif` statements in the code, we separate the dataset of field length 5 and the dataset of field length 2 being *Contracts* and the *Job 1* output respectively. For each of these datasets, we set the join keys to the `to_address` and `address` but set the join values as `block_number` in *Contracts* and `total_value` in the *Job 1* output. In both statements, we yield the join key with its the corresponding join value and the number 1 or 2 if it belongs to *Contracts* or *Job 1* output respectively. We pass all the irrelevant information.

Reducer

```
    def reducer(self, address, values):

        val = ['','']

        for value in values:
            if value[1] == 1:
                val[0] = value[0]
            elif value[1] == 2:
                val[1] = value[0]

        if val[0] != '':
            yield (address, val[1])
```

This is where we now implement our join. Firstly, we create an empty array with two blank positions. We loop over all the values from the output of our mapper parsed to the reducer as `values`. If the second position in a particular value is 1, it means it belongs to *Contracts* and we will store the corresponding join value in first position of our empty array. Similarly, if this value is 2, it belongs to the output of *Job 1* and we store the corresponding join value into the second position of our empty array. This will now transform the empty array into one with many rows, some rows with one empty input and some with none. We only yield the address (i.e. the join key) together with the second position (i.e. the `total_value` of an address) of the rows with no empty positions, since we only want to consider the smart contracts which belong in the contracts.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574171293853_6016/

JOB 3 - TOP TEN

Finally, this job will take the now filtered address aggregates and take a top ten of the smart contracts by the total Ether received. Here are these top ten smart contracts in a table:

Address of Contract	Total Ether Received
1. 0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444	84155100809965865822726776
2. 0xfa52274dd61e1643d2205169732f29114bc240b3	45787484483189352986478805
3. 0x7727e5113d1d161373623e5f49fd568b4f543a9e	45620624001350712557268573
4. 0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef	43170356092262468919298969
5. 0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8	27068921582019542499882877
6. 0xbfc39b6f805a9e40e77291aff27aee3c96915bdd	21104195138093660050000000
7. 0xe94b04a0fed112f3664e45adb2b8915693dd5ff3	15562398956802112254719409
8. 0xbb9bc244d798123fde783fcc1c72d3bb8c189413	11983608729202893846818681
9. 0xabbb6bebfa05aa13e908eaa492bd7a8343760477	11706457177940895521770404
10. 0x341e790174e3a4d35b65fdc067b6b5634a61caea	8379000751917755624057500

Here are the MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            if len(line.split('\t')) == 2:
                line = line.strip()
                fields = line.split('\t')
                address = fields[0]
                val = int(fields[1])
                yield (None, (address, val))
        except:
            pass
```

Similarly to the mapper in *Part A*, we split the lines into fields but where they are being tab-separated since I saved the output of *Job 2* as a `.tsv` file. We consider only those inputs with field length 2. We yield the value `None`, since we only require the tuple we are yielding this with being the `address` and `total_value` fields of our input. Again, we pass all the irrelevant information. The output of the mapper will be unsorted so it is the job of the reducer to sort them in order to obtain a top ten.

Combiner

```
    def combiner(self, _, unsorted_values):
        sorted_values = sorted(unsorted_values, reverse=True, key = lambda
unsorted_vals:unsorted_vals[1])
        i=0
        for value in sorted_values:
            yield("top", value)
            i += 1
            if i >= 10:
                break
```

Introducing a combiner will help reduce the workload of the reducer by summarising or "combining" some of mapper output. We take the same input as our reducer would, which is the output of the mapper. We sort the values in descending numerical order with respect to the `total_value` field. Next, we yield the top `value` in the `sorted_values`, and we stop until we reach the tenth biggest total. The combiner will yield the top ten smart contract addresses but with a group of values which will need to be reduced.

Reducer

```
    def reducer(self, _, unsorted_values):
        sorted_values = sorted(unsorted_values, reverse=True, key = lambda
tup:tup[1])
        i=0
        for value in sorted_values:
            yield("{} - {}".format(value[0], value[1]), None)
            i += 1
            if i >= 10:
                break
```

We now take the output of the combiner and we do the same operations as in the combiner except we yield the top ten smart contract addresses but now with their overall totals.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574171293853_6051/

PART C. DATA EXPLORATION (30%)

1. GAS GUZZLERS

JOB 1 - GAS PRICE OVER TIME

For any transaction on Ethereum, a user must supply gas. Here is a bar plot showing the average gas price for each month between the start and end of the *Transactions* dataset provided in HDFS:

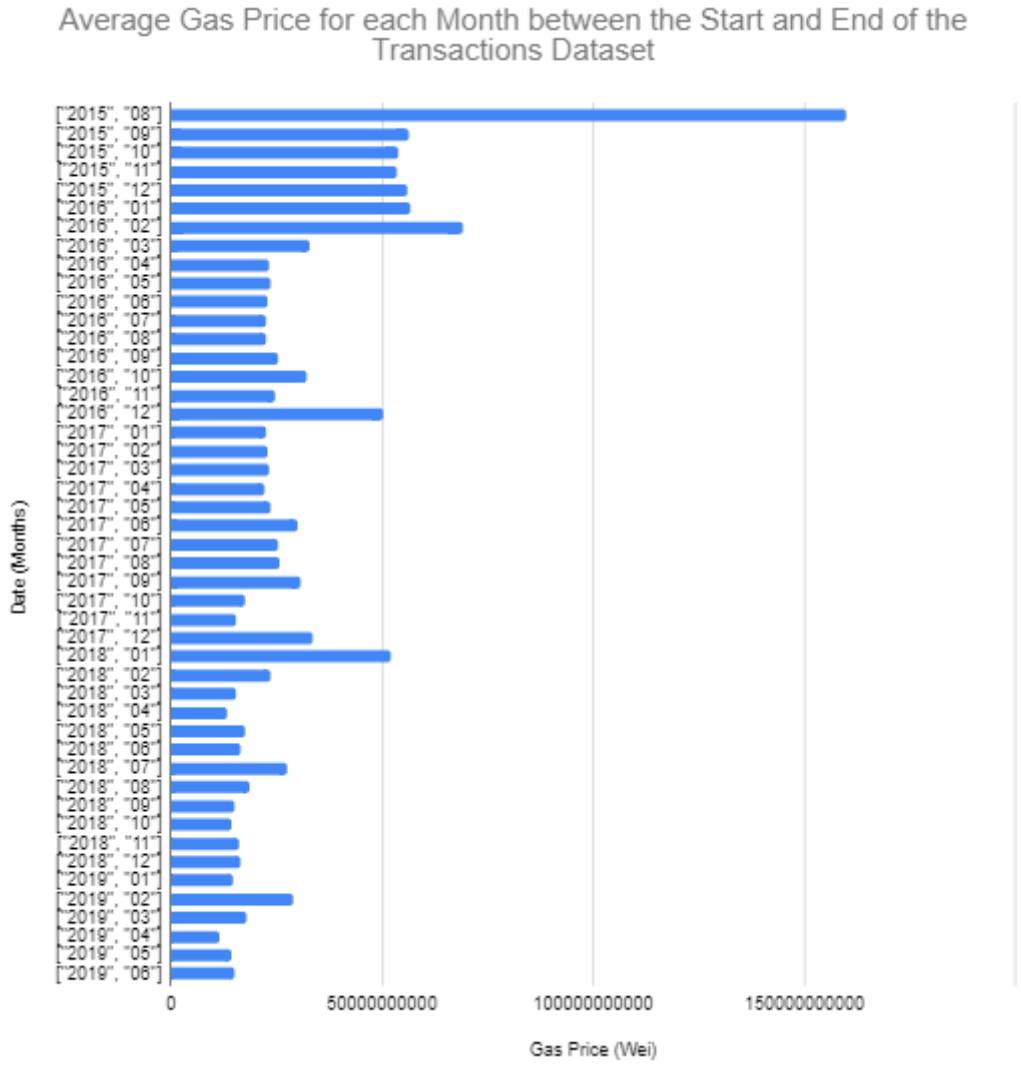

Figure 2: A bar plot showing the average price for each month of the transactions dataset.

We see a very high gas price at the start of the dataset with a rapid decline in the first month and after this, we observe the gradual and fluctuating decline of the gas price.

Here are the MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            fields = line.split(",")
            if len(fields) == 7:
                time_epoch = int(fields[6])
                month = time.strftime("%m",time.gmtime(time_epoch))
                year = time.strftime("%Y",time.gmtime(time_epoch))
                key = (year, month)
                gas_price = int(fields[5])
                yield (key, (gas_price, 1))
        except:
            pass
```

Similar to *Part A*, we take the `/data/ethereum/transactions` dataset as our input and split the lines into fields where they are being comma separated for those inputs which have precisely 7 fields. We need to convert the `block_timestamp` field from time epoch to using the `strftime` operation. We store the year and month into a key which we yield along with the `gas_price` with a count 1.

Combiner

```
    def combiner(self, key, gas_price_sum):
        gas_price_total = 0
        count_total = 0
        for n in gas_price_sum:
            gas_price_total += int(n[0])
            count_total += int(n[1])
        yield (key, (gas_price_total, count_total))
```

Again, we introduce a combiner to reduce the workload of the reducer. We create variables `gas_price_total` and `count_total` both equal to 0. We then add all the gas prices and counts using a `for` loop. We then yield the key as defined in the mapper along with a tuple consisting of the `gas_price_total` and `count_total`.

Reducer

```
    def reducer(self, key, gas_price_sum):
        gas_price_total = 0
        count_total = 0
        for n in gas_price_sum:
            gas_price_total += int(n[0])
            count_total += int(n[1])
        yield (key, gas_price_total / count_total)
```

We have the same inputs as the combiner being the key and the tuple except that the combiner has grouped the keys with the sum of the gas prices and counts. The reducer will now perform the same operations as in the combiner but instead will yield the key with `gas_price_total` divided by `count_total` to perform the calculation of the average gas price for each month.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574171293853_6554/

JOB 2A - HAVE CONTRACTS BECOME MORE COMPLICATED? : REPLICATION JOIN

To see how complicated or not contracts have become, we will look at how average monthly gas use of all the top ten contract transactions from *PartB* changes. Additionally, we will compare this with the average monthly value of Ethereum transferred for the top ten contract transactions, to see if there is a correlation to be found. Here are two bar plots presenting this:

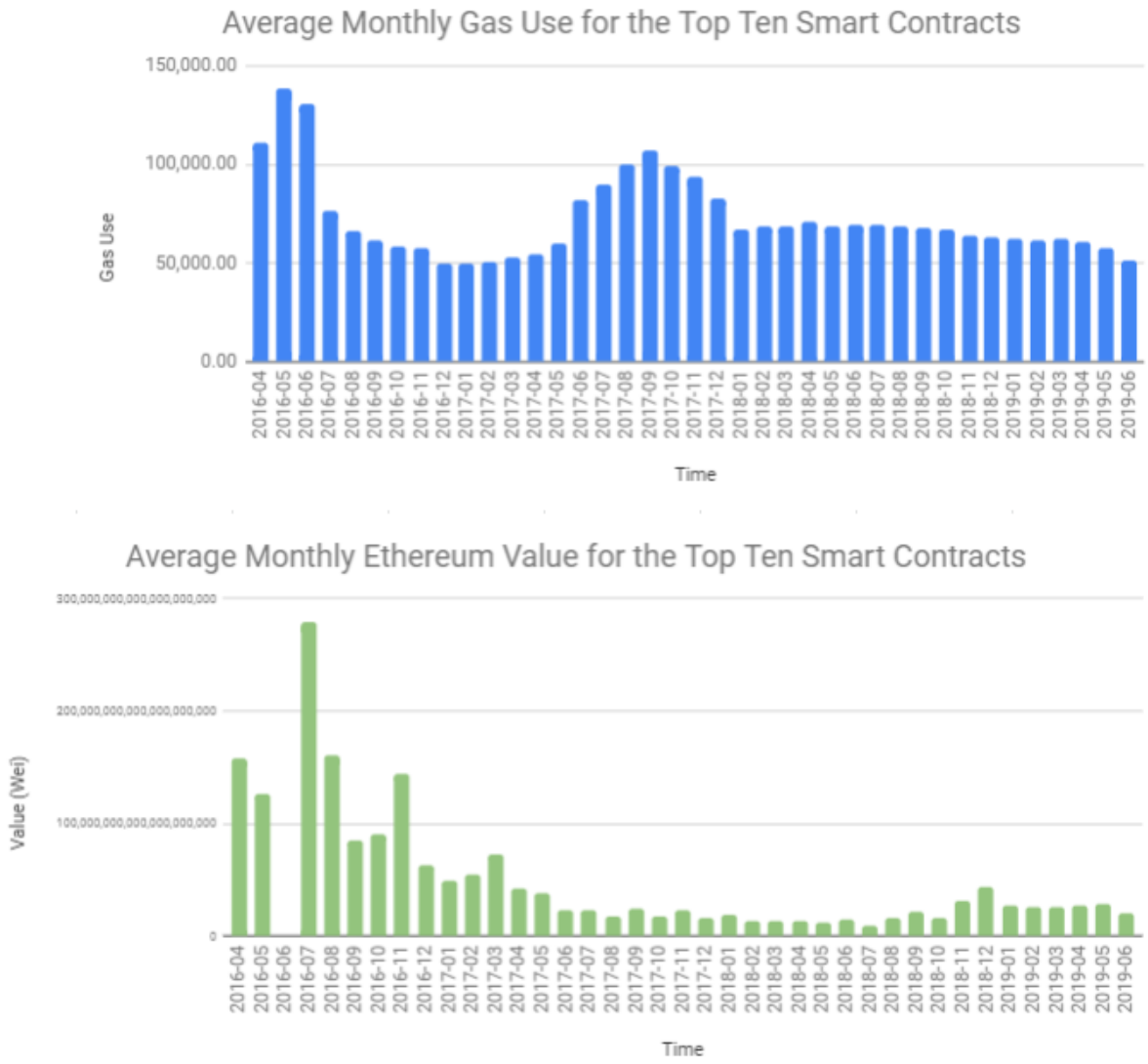

Figure 2 (top): A bar plot showing the gas use of the top ten contract transactions over each month.

Figure 3 (bottom): A bar plot showing the value transferred of the top ten contract transactions for each month.

We can see that the value and gas use both begin with higher values and get smaller over time. However, the gas use increases again when reach mid/late 2017 and drops at the start of 2018 where it remains at a steady 50,000. The value is generally decreasing with some small fluctuations from late 2018 onwards. We may be able to see some general correlation between the value and gas use in the year 2016 but it is slightly more ambiguous in later years to draw a connection and it would be reasonable to think there could be other influencing factors involved. We can draw a stronger correlation with the value in *Figure 3* and the gas value in *Figure 2*.

To achieve the results shown, we will perform a replication join with the top ten contracts and the *Transactions* table and then an aggregate of both the gas and values fields.

Here are the MapReduce steps involved in the replication join:

Mapper

```
    contracts = {}

    def mapper_join_init(self):
        with open("PartBTopTen.tsv") as f:
            for line in f:
                fields = line.split("\t")
                key = fields[0]
                val = fields[1]
                self.contracts[key] = val

    def mapper_repl_join(self, _, line):
        fields = line.split(",")

        for i in self.contracts:
            to_address = fields[2]

            if to_address == i:
                value = fields[3]
                gas = fields[4]
                time_epoch = int(fields[6])
                time_t = time.strftime("%Y-%m", time.gmtime(time_epoch))
                tup = (time_t, gas, value)
                yield (to_address, tup)
                break
            else:
                pass

    def mapper_length(self, key, values):
        yield(key, list(values))
```

In the mapper, we perform a map-side join. We first split the input being the top ten contracts and set the contract address as the join key and total value as the join value and store it in the global variable `contracts` which we can call later. Next, we want to join it with the *Transactions* and we only join if the `to_address` is equal to the `address` of a top ten contract transaction. We yield the join key of the *Transactions* being the `to_address` along with the tuple consisting of the time (which has been converted from epoch to year and month), the gas and the value. We pass all the other `to_address` values as they are not part of the top ten contract transactions. Lastly, we yield the join key with a list of the join values stated in the tuple.

Reduce

```
    def reducer_sum(self, key, values):
        for val in values:
            yield (key, (val[0], val[1], val[2]))

    def steps(self):
        return [MRStep(mapper_init=self.mapper_join_init,
                       mapper=self.mapper_repl_join),
                MRStep(mapper=self.mapper_length,
                       reducer=self.reducer_sum)]
```

We then yield in our reducer the join key along with the 3 join values. We need to use the steps function from `mrjob` we imported to be able to run the previous MapReduce jobs one after the other.

After these steps, we get an output of all the transactions belonging to a top ten contract, with a date of the transaction, the gas used and the value.

JOB 2B - HAVE CONTRACTS BECOME MORE COMPLICATED? : GAS AND VALUE AGGREGATION WITH TIME

Now we will aggregate of both the gas and values fields to obtain the desired output.

Here are the MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            fields = line.split("\t")
            if len(fields) == 2:
                fields2 = fields[1].split(',')
                date = fields2[0][2:9]
                gas = fields2[1].replace('"', '')
                value = fields2[2][2:].replace('"]', '')

                yield (date, (gas, value, 1))
        except:
            pass
```

First, we need to filter the output of the previous jobs. We do two splits as we have a tab-separated part and a comma-separated part of the output where we used a join key and 3 join values. Our input has only 2 fields which are tab-separated so we only yield if this condition is met. We then remove any `"` or `[]` and then yield the date along with the gas, the value and a count of 1 as we want to aggregate on the gas and value. We pass all the irrelevant information.

Combiner

```
    def combiner(self, key, sum):
        value_total = 0
        gas_total = 0
        count_total = 0
        for n in sum:
            gas_total += int(n[0])
            value_total += int(n[1])
            count_total += int(n[2])
        yield (key, (gas_total, value_total, count_total))
```

Again, we use a combiner to reduce the workload of the reducer. We create variables `gas_total`, `value_total` and `count_total` all equal to 0. We then add all the gas, values and counts using a `for` loop. We then yield the key as defined in the mapper along with a tuple consisting of the `gas_total`, `value_total` and `count_total`.

Reducer

```
    def reducer(self, key, sum):
        value_total = 0
        gas_total = 0
        count_total = 0
        for n in sum:
            gas_total += int(n[0])
            value_total += int(n[1])
            count_total += int(n[2])
        yield (key, (gas_total / count_total, value_total / count_total))
```

We have the same inputs as the combiner being the key and the tuple except that the combiner has grouped the keys with the sum of the gas, values and counts. The reducer will now perform the same operations as in the combiner but instead will yield the key with `gas_total` divided by `count_total` as well as the `value_total` divided by `count_total` to perform the calculation of the average gas use and value for each month.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_5987/

2. SCAM ANALYSIS

JOB 1A - MOST LUCRATIVE FORM OF SCAM: REPARTITION JOIN

We can utilise the provided scam dataset to find which form of scam is the most lucrative form of scam. Here are the results in the following table:

Form of Scam	Overall Value (Wei)
1. Scamming	36580114911015548568219
2. Phishing	26755326559607588846701
3. Fake ICO	205154541098712454112

I first convert the JSON file to a CSV so the *scams.json* file is can be joined with the *PartBJob1* output. We will then join the *Scams* and the *PartBJob1* output using the following MapReduce steps:

Mapper

```
    def mapper(self, _, line):
        try:
            if len(line.split(','))==37:
                line = line.strip()
                fields = line.split(',')
                join_key = fields[1]      #join_key is scam_address
                join_key = join_key.replace('"', '')
                join_value = fields[6]      # join_value is category
                yield (join_key, (join_value,1)) # 1 for scams
```

```
            elif len(line.split('\t'))==2:
                line = line.strip()
                fields = line.split('\t')
                join_key = fields[0]                #join key is to_address
                join_value = int(fields[1])       #join_value is   value sums
                join_key = join_key.replace('"', '')
                yield (join_key,(join_value,2)) # 2 for JOB1 OUTPUT

        except:
            pass
```

We take the *PartBJob1* output and *Scams.csv* datasets as our inputs and split the lines into fields where they are being tab/comma separated. We introduce `if` and `elif` statements in the code and we separate the dataset of field length 37 and the dataset of field length 2 being *Scams.csv* and the *PartBJob1* output respectively. For each of these datasets, we set the join keys to the `scam_address` and `to_address` but set the join values as `category` in *Scams.csv* and `total_value` in the *ParBJob1* output. In both statements, we yield the join key with its the corresponding join value and the number 1 or 2 if it belongs to *Scams.csv* or *PartBJob1* output respectively. We pass all the irrelevant information.

Reducer

```
    def reducer(self, address, values):
        category = ''
        val = ''

        for value in values:
            if value[1] == 1:
                category = value[0]
            elif value[1] == 2:
                val = value[0]

        if category != '' and val !='':
            yield (category, val)
```

In the reducer, we implement the join. Firstly, we create to empty variables `category` and `val`. We loop over all the values from the output of our mapper parsed to the reducer as `values`. If the second position in a particular value is 1, it means it belongs to *Scams* and we will store the corresponding join value in `category`. Similarly, if this value is 2, it belongs to the output of *PartBJob1* and we store the corresponding join value into `val`. We now yield the `category` with the `val` only if a row in `category` and `val` are both not empty.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_1499/

JOB 1B - MOST LUCRATIVE FORM OF SCAM: AGGREGATION

We shall perform an aggregation on the output of the previous job, being a transactions labelled by their scam form with a value of the transaction, to obtain the overall value of each scam type.

Here are the following MapReduce steps involved:

Mapper

```
    def mapper(self, _, line):
        try:
            fields = line.split("\t")
            if len(fields) == 2:
                category = fields[0]
                value = int(fields[1])
                yield (category, value)
        except:
            pass
```

Similarly to previous jobs, we will split the table into fields where it is tab-separated and only consider the inputs with fields length 2. We then yield the `category` along with the field we want to aggregate on, being the `value`. We pass all the irrelevant information.

Reducer

```
    def reducer(self, category, value):
        yield (category, sum(value))
```

We now yield the sum of all the values for each distinct category to obtain results in the aforementioned table.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_1620/

JOB 2 - HOW SCAM CATEGORIES LUCRATIVENESS CHANGE OVER TIME?

We now would like to see when the different categories were most lucrative and when they were not so successful. Here is the following line graph illustrating this:

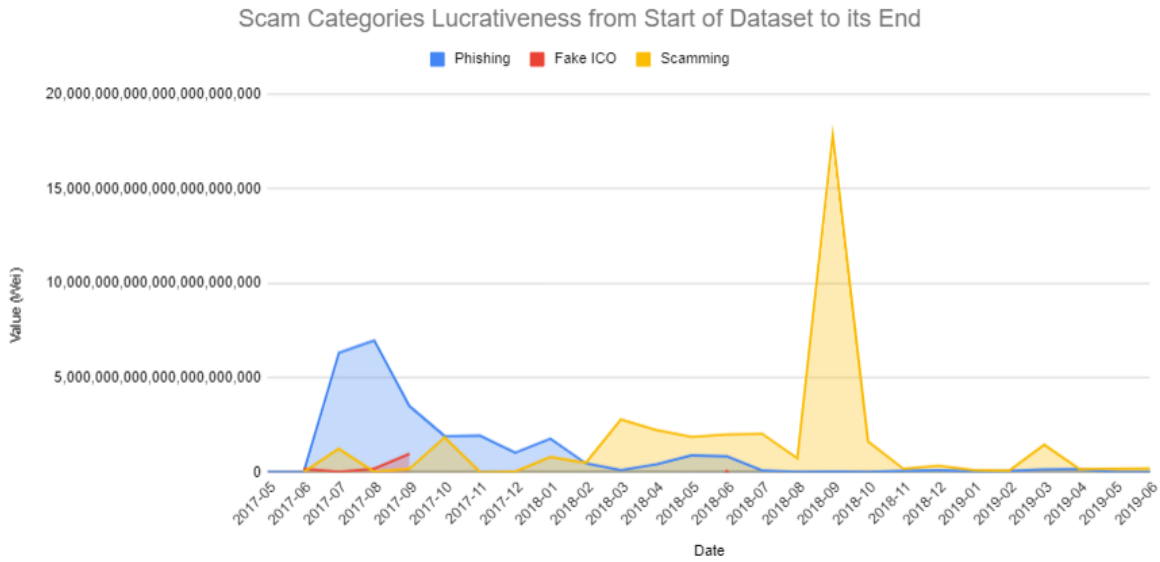

Figure 4: A line chart showing the scam categories lucrativeness over time.

From observation of the chart, we can see that in 2017 *Phishing* was had the most gain, with only very low Wei gains from the other two categories. An article (https://blog.alertlogic.com/must-know-phishing-statistics-2018/) here tells us that 76% of organisations said that they experienced phishing attacks in 2017. WannaCry in particular, a ransomware attack, was made pretty much subsided in around June 2017 and perhaps explaining why there was a decline after this point.

Fake ICO is essentially non-existent after 2017-09 with an occurrence in 2018-06. After 2018-02, *Scamming* becomes the most lucrative scam type with a value skyrocketing to around $20 * 10^{21}$ Wei. It is likely that this particular month is the cause for it being the top scam type in *Job1*. After 2018-11, most scam types have taken little Wei.

To provide the values for the chart in *Figure 4*, we need to perform a replication join with the `scams.json` file and the `Transactions` dataset.

Here are the MapReduce steps involved:

Mapper

```
    scam = {}

    def mapper_join_init(self):
        json_file = io.open("scams.json", 'r', encoding='utf-8-sig').read()
        p_json = json.loads(json_file)
        result = p_json['result']
        for r in result:
            self.scam[r] = result[r]['category']

    def mapper_repl_join(self, _, line):
        fields = line.split(",")
        for i in self.scam:
            to_address = fields[2]

            if to_address == i:
                value = int(fields[3])
                time_epoch = fields[6]
                date = time.strftime("%Y-%m", time.gmtime(int(time_epoch)))
                yield ((date, self.scam[i]), value)
                break
            else:
                pass

    def mapper_length(self, key, values):
        yield(key, values)
```

I was able to find a tool to import to access the JSON file in the I implemented map-side replication join which is defined in `mapper_join_init`. We then join this with the *Transactions* on the join key `to_address` for all the scam address present in the `scam_json` file. We then yield the date and scam addresses present in the *Transactions* dataset along with the transaction value. We pass all the other `to_address` values as they are not scams. Lastly, we yield the join key with a list of the join values stated in the tuple.

Reducer

```
    def reducer_sum(self, key, values):
            yield (key, sum(values))

    def steps(self):
        return [MRStep(mapper_init=self.mapper_join_init,
                       mapper=self.mapper_repl_join),
                MRStep(mapper=self.mapper_length,
                       reducer=self.reducer_sum)]
```

Similarly to *PartC Job2B*, we yield in our reducer the join key along with the sum of the values, so that we get the monthly totals. Again, we need to use the steps function from `mrjob` we imported to be able to run the previous MapReduce jobs one after the other.

Here is the corresponding id for this job
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1574975221160_5494/