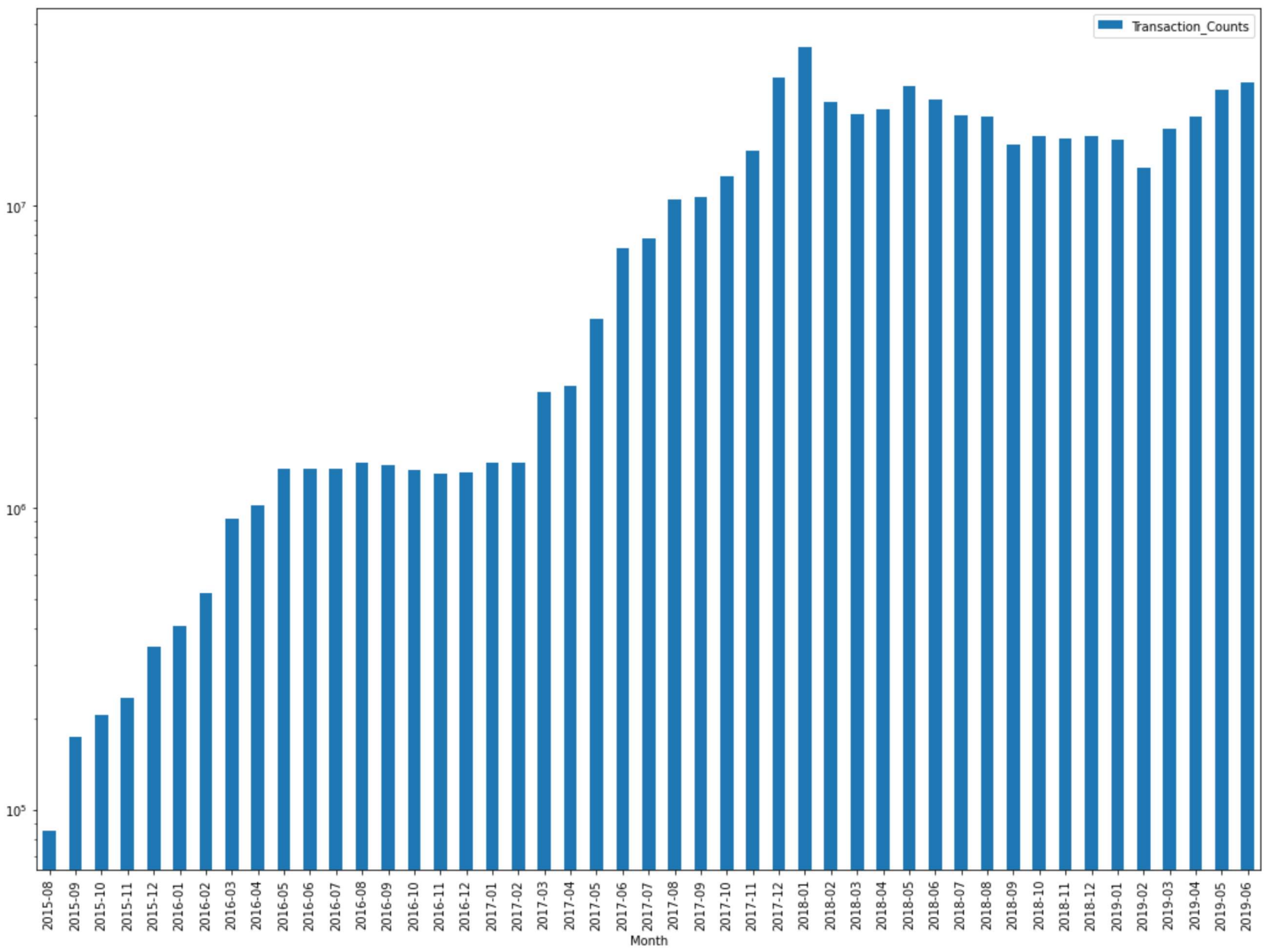
Part A

Ethereum Time Analysis: Number of Transactions per Month

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_2055/

In [ ]:
```
from mrjob.job import MRJob
from datetime import datetime


class ETH_Times(MRJob):

    # Sorts the input to the reducers, only (and not the merged output too)
    MRJob.SORT_VALUES = True

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

        try:
            if len(fields) == 7:
                block_timestamp = int(fields[6])

                ## block_timestamp is converted to date format yyyy-MM, from POSIX timestamp
                part_date = datetime.fromtimestamp(block_timestamp).strftime('%Y-%m')

                ## The partial date (part_date) yyyy-MM is the key,
                ## while the value yielded is '1', given this is a counting job
                yield part_date, 1
        except:
            pass

    # Use of combiner to increase efficiency
    def combiner(self, part_date, counts):
        yield part_date, sum(counts)

    def reducer(self, part_date, counts):
        yield part_date, sum(counts)


if __name__ == '__main__':

    # For more than 1 reducer, the merged output is NOT sorted, although complete
    ETH_Times.JOBCONF = {'mapreduce.job.reduces': '1'}
    ETH_Times.run()
```

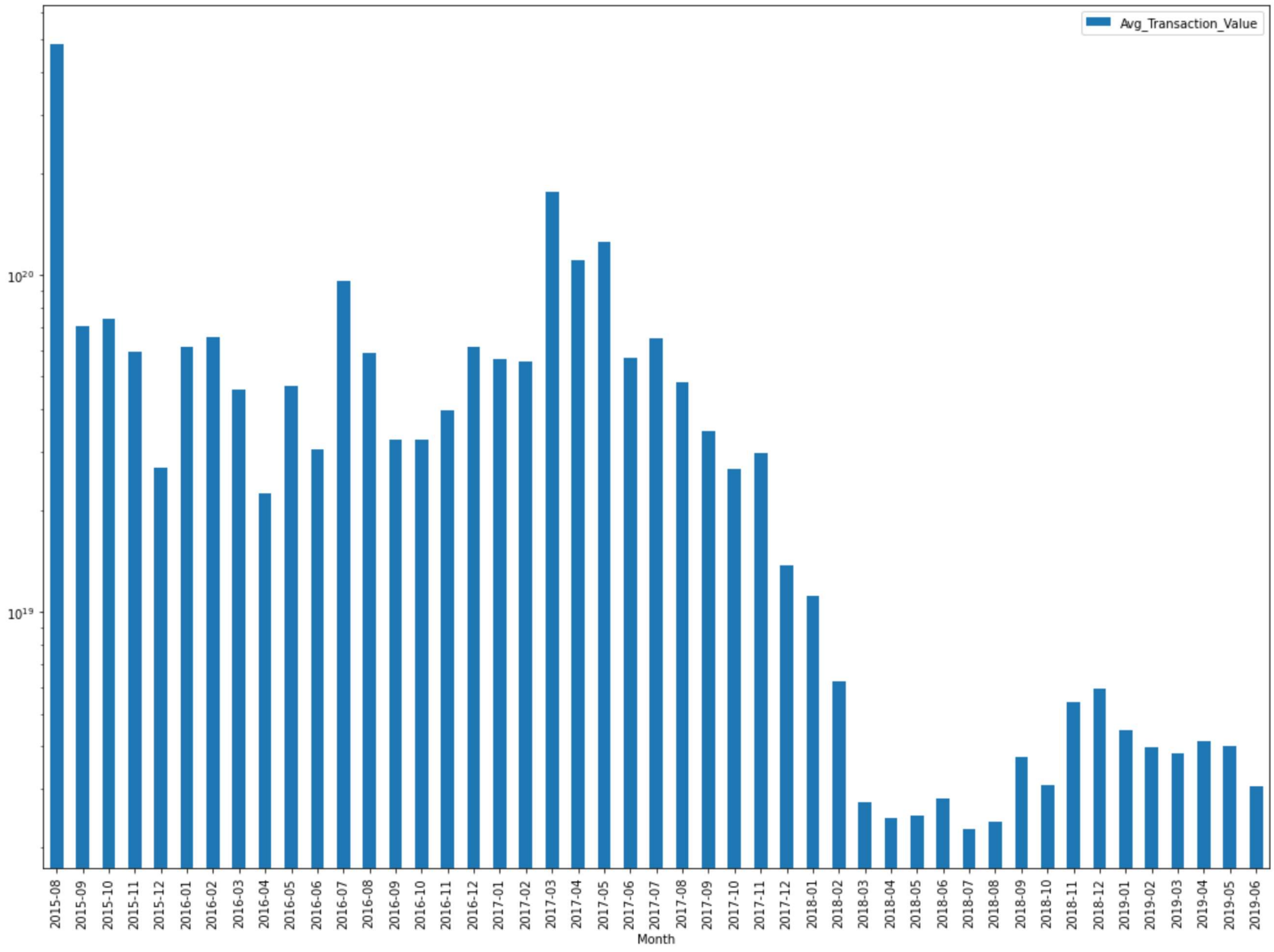


Ethereum Time Analysis: Average Value of Transactions per Month

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_2056/

In [ ]:
```
from mrjob.job import MRJob
from datetime import datetime


class ETH_AvgTimes(MRJob):

    # Sorts the input to the reducers, only (and not the merged output too)
    MRJob.SORT_VALUES = True

    def mapper(self, _, line):
        fields = line.split(",")
        try:
            if len(fields) == 7:
                block_timestamp = int(fields[6])

                ## block_timestamp is converted to date format yyyy-MM, from POSIX timestamp
                part_date = datetime.fromtimestamp(block_timestamp).strftime('%Y-%m')

                transaction_value = int(fields[3])
```

```
                ## The partial date (part_date) yyyy-MM is the key.
                ## On this occasion, transaction_value is paired with '1' to facilitate counting,
                ## given this job is tasked with computing the average of all transaction values per part_date
                yield part_date, (transaction_value, 1)
        except:
            pass

    def combiner(self, part_date, pairs):

        ## For a key (i.e., part_date), compute its local (i.e., from a mapper) average components: sum & counts
        ## Given that other mappers may have the same part_date key, the average is not computed yet.
        yield part_date, self.feature_counter(pairs)

    def reducer(self, part_date, pairs):

        ## Same as above, except that, on this occasion, the components will be plugged into the average formula
        (transaction_values, counts) = self.feature_counter(pairs)

        ## Computing the average for a key (i.e., part_date)
        yield part_date, transaction_values/counts

    def feature_counter(self, pairs):
        (transaction_values, counts) = (0, 0)

        ## For a key, as seen in the combiner/reducer:
        ## add up all its corresponding collected values (i.e., transaction_value), and
        ## add up the counts for how many transactions have been collected (so far, if in combiner) in that year & month
        for (transaction_value, count) in pairs:
            transaction_values += transaction_value
            counts += count

        return transaction_values, counts


if __name__ == '__main__':

    # For more than 1 reducer, the merged output is NOT sorted, although complete
    ETH_AvgTimes.JOBCONF = {'mapreduce.job.reduces': '1'}
    ETH_AvgTimes.run()
```



Part B. TOP TEN MOST POPULAR SERVICES

JOB 1 - INITIAL AGGREGATION

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_1990/

In [ ]:

```
from mrjob.job import MRJob
from mrjob.step import MRStep


class ETH_TopServices(MRJob):

    ####
    ###
    # JOB 1 - INITIAL AGGREGATION
    ###
    ####
    def transactions_filter(self, _, line):
        fields = line.split(",")

        try:
            if len(fields) == 7:

                to_address = fields[2]
                value = int(fields[3])

                ## For aggregating the total value of all transactions per 'to_address',
                ## it is neccessary to yield the 'to_address' field as a key, and
                ## the transaction 'value' field it corresponds to.
```

```
                yield to_address, value
        except:
            pass

    ## Computing (locally - i.e., for the keys in 1 mapper)
    ## the sum of all transaction values mapped to a given key (i.e., to_address)
    def transactions_agg_combiner(self, to_address, value):
        yield to_address, sum(value)

    ## Same as above but computing the sum of sums computed by combiners,
    ## then yielding the final output.
    def transactions_agg_reducer(self, to_address, value):
        yield to_address, sum(value)

    def steps(self):
        return [MRStep(mapper   = self.transactions_filter,
                       combiner = self.transactions_agg_combiner,
                       reducer  = self.transactions_agg_reducer)]

if __name__ == '__main__':
    ETH_TopServices.JOBCONF = {'mapreduce.job.reduces': '35'}
    ETH_TopServices.run()
```

JOBS 2 & 3

JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_1998/

JOB 3 - TOP TEN

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_1999/

In [ ]:

```
from mrjob.job import MRJob
from mrjob.step import MRStep

class ETH_TopServices(MRJob):

    ####
    ###
    # JOB 2 - JOINING TRANSACTIONS/CONTRACTS AND FILTERING
    ###
    ####

    ## It is essential to tell apart the datasets to be joined.
    ## Thus, the to_address fields and their corresponding aggregate
    ## received from JOB 1 are tab-separated ('\t') and are tuples of 2 per line (i.e., length of fields is 2),
    ## whereas the 'Contracts' dataset is comma-separated and has 5 fields per line.
    def repartition_filter(self, _, line):
        try:
            ### From JOB 1, aggregated transaction values grouped by to_address
            if len(line.split('\t')) == 2:
                fields = line.split('\t')

                ## Each key is surrounded by a set of quotes in the tab-separated file.
                ## Without stripping these away, mrjob's encoding would attempt
                ## to surround these with another set of quotes.
                ## Eventually, failing to recognise them. Thus, it is essential to remove these...
                join_key = fields[0].strip('"')

                ## The value paired with a key although evidently a big integer,
                ## it is essential to convert an propagate this as an actual integer, not as a string.
                join_value = int(fields[1])
                yield join_key, join_value

            else:
                ### From the 'Contracts' dataset
                if len(line.split(',')) == 5:
                    fields = line.split(',')

                    ## In order to identify smart contract addresses within the to_address field from above,
                    ## we must send the the contract addresses as keys, along with a "tag".
                    ## In reducers, the keys mapped to the nominated "tag" are certain to be contract addresses,
                    ## as those keys were paired with this tag in else-if branch, which is associated with the
                    ## the 'Contracts' dataset.
                    join_key = fields[0]
                    ## The nominated tag is "contract".
                    yield join_key, "contract"
        except:
            pass

    def repartition_reducer(self, contract_addr, joined_values):


        ###
        ## If joined_values mapped to a given key contains the "contract" tag, and
        ##  it also contains a numerical value (the total value of transactions for a key),
        ##  then the address (i.e., the key) is a smart contract address,
        ##  which can be found in both datasets.
        ## Hence, in the tab-separated file, this address exists and it is that of a smart contract.
        ###
        ## There could be multiple "contract" tags for a given key,
        ##  if that given key (i.e., address) appears multiple times within the 'Contracts' dataset.
        ###

        ## We shall assume that there are no numerical values or that there are no tags
        contract = False
        contract_value = None

        for jv in joined_values:

            if jv != "contract":
            ## Finding the numerical value (i.e., the total value of transactions made to to_address - the key)
                contract_value = jv ## This line must be executed EXACTLY ONCE
                ## (i.e., only 1 such numerical value - an aggregate)
            else:
            ## Determining whether the address is that of a smart contract
                contract = True ## This line must be executed AT LEAST ONCE
                ## (e.g., multiple addresses in the 'Contracts' dataset)

        ## There could be cases where the address appears only in the 'Contracts' dataset,
        ##  which implies there is no numerical value in 'joined_value' and only (1 or more) "contract" tags.
        ## Also, there could be cases in which there is only a numerical value and no tags,
        ##  which means that the address is not that of a smart contract.
        ## We will filter out such cases by checking
        ##  whether we found the right criteria -- using 'contract':bool and 'contract_value':bigint variables.
        ## If the address is that of a smart contract, then the address and its corresponding total value of transactions
```

```
        ##  will form a tuple and this tuple will then be sent to a single (executing) reducer for sorting and listing.
        if (contract_value is not None) and (contract is True):
            yield "top10", (contract_addr, contract_value)

    ####
    ###
    # JOB 3 - TOP TEN
    ###
    ####

    ## After receiving all the tuples from the previous reducers of "JOB 2 - JOINING TR..." (i.e., via key "top10"),
    ## the single (executing) sorting reducer will sort in descending order the aggregated addresses
    ## by their corresponding paired values.
    ## It will then yield as final output only the first 10 records.
    ## Hence, the top 10 services (i.e., contracts) by the total value of (received) transactions.
    def get_top10_services(self, _, contract_data):
        top10 = sorted(contract_data, reverse=True, key=lambda tup: tup[1])
        top10 = top10[:10]

        for record in top10:
            yield record[0], record[1]

    def steps(self):
        return [MRStep(mapper  = self.repartition_filter, reducer = self.repartition_reducer),
                MRStep(reducer = self.get_top10_services)]


if __name__ == '__main__':
    ETH_TopServices.JOBCONF = {'mapreduce.job.reduces': '35'}
    ETH_TopServices.run()
```

Final Output: Contract Address v. Total Contract Value (in Wei)

```
"0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444"    84155100809965865822726776
"0xfa52274dd61e1643d2205169732f29114bc240b3"    45787484483189352986478805
"0x7727e5113d1d161373623e5f49fd568b4f543a9e"    45620624001350712557268573
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef"    43170356092262468919298969
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8"    27068921582019542499882877
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd"    21104195138093660050000000
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3"    15562398956802112254719409
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413"    11983608729202893846818681
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477"    11706457177940895521770404
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"    83790007519177556240575000
```

PART C. TOP TEN MOST ACTIVE MINERS

```
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_2107/
http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_2108/
```

In [ ]:

```
from mrjob.job import MRJob
from mrjob.step import MRStep

### Executed on 'Blocks' dataset
class ETH_TopMiners(MRJob):

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

        try:
            ## Dismissing malformed lines (i.e., with less or more than 9 fields after splitting by ',')
            if len(fields) == 9:

                miner_addr = fields[2]
                block_size = int(fields[4])

                yield miner_addr, block_size
        except:
            pass

    def agg_combiner(self, miner_addr, block_size):

        ## Aggregating the miner using its address, miner_addr, and
        ## yielding the sum of all its corresponding mined block sizes
        yield miner_addr, sum(block_size)

    def agg_reducer(self, miner_addr, block_size):

        ## Similar with above, but the key is constant ("aggregated").
        ## This will redirect the output of this job's reduces to a single reducer from a secondary MR job.
        ## That 1 reducer will receive, as a tuple (i.e., pair):
        ## all collected and aggregated miner addresses, along with total size of mined blocks sizes.
        yield "aggregated", (miner_addr, sum(block_size))

    ## After receiving all the tuples from the previous reducers of the first MR job (i.e., via key "aggregated"),
    ## the single (executing) sorting reducer will sort in descending order the aggregated addresses
    ## by the total size of mined blocks (i.e., their corresponding paired values).
    ## It will then yield as final output only the first 10 records.
    ## Hence, the top 10 miners by the size of blocks mined.
    def sorting_reducer(self, _, pairs):
        top10_miners = sorted(pairs, reverse=True, key=lambda tup: tup[1])
        top10_miners = top10_miners[:10]

        for record in top10_miners:
            yield record[0], record[1]

    def steps(self):
        return [MRStep(mapper=self.input_filter, combiner=self.agg_combiner, reducer=self.agg_reducer),
                MRStep(reducer=self.sorting_reducer)]


if __name__ == '__main__':
    ETH_TopMiners.run()
```

Final Output: Miner Address v. Total Size of Blocks Mined

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

PART D. DATA EXPLORATION

SCAM ANALYSIS

JSON Parser for *Popular Scams*

In [ ]:
```
import json as jsn

scams = {}

def filter():
    with open("scams.json") as f:
        scams_data = jsn.load(f)

        for k, v in scams_data['result'].items():

            id = str(v['id'])
            category = v['category'].lower()
            status = v['status'].lower()

            ## Because there were many records with the same scam-report 'id',
            ## assigning in a dictionary to the same key (i.e., considered this 'id'),
            ## would avoid duplicates
            scams[id] = {'category': category, 'status': status}
filter()

## A duplicates-free dictionary is then printed in CSV-format to a file
## (e.g., this_code.py > output.csv)
for k, v in scams.items():
    print(f"{k},{scams[k]['category']},{scams[k]['status']}")
```

Filtered Scam Types to CSV file for *Popular Scams*

```
130,phishing,offline
1200,phishing,active
6,phishing,offline
...
```

Popular Scams *by frequency*

http://andromeda.student.eecs.qmul.ac.uk:8088/proxy/application_1608030573468_1935/

In [ ]:
```
from mrjob.job import MRJob
from mrjob.step import MRStep


class ETH_PopularScams(MRJob):

    def scam_category_mapper(self, _, line):
        fields = line.split(",")
        try:
            if len(fields) == 3:
                category = fields[1]

                ## Similar as in previous discussions, this is a counting job for each category.
                yield category, 1
        except:
            pass

    ## Similar as in previous discussions, this is a counting job for each category.
    def scam_category_combiner(self, category, count):
        yield category, sum(count)

    ## Similar as in previous discussions,
    ## the tuples are sent for sorting to 1 separate reducer in a secondary job.
    ## Constant key is "popular" on this occasion.
    def scam_category_reducer(self, category, count):
        yield "popular", (category, sum(count))

    ## The single (executing) reducer sorts in descending order, and
    ## lists the top 10 categories it has sorted.
    def list_sort_categories(self, _, pairs):
        sorted_categories = sorted(pairs, reverse=True, key=lambda x: x[1])

        yield f"Most popular type of scam, with {sorted_categories[0][1]} counts: ", sorted_categories[0][0]

        for tup in sorted_categories:
            yield tup[0], tup[1]

    def steps(self):
        return [MRStep(mapper=self.scam_category_mapper, combiner=self.scam_category_combiner,
                       reducer=self.scam_category_reducer),
                MRStep(reducer=self.list_sort_categories)]


if __name__ == '__main__':
    ETH_PopularScams.run()
```

Final Output for *Popular Scams*:

```
"Most popular type of scam, with 1460 counts: " "scamming"
"scamming"  1460
"phishing"  517
"fake ico"  5
"scam"  1
```

Most Lucrative Scams *by value of transactions*

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1608030573468_1973
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1608030573468_1974

In [ ]:
```
from mrjob.job import MRJob
from mrjob.step import MRStep
import json as jsn

## To be executed on the 'Transactions' dataset.
class ETH_LucativeScams(MRJob):

    scams = {}

    ## addr_filter() is related to mapper_init for the first job and reducer_init for the second.
    ## This is executed before their corresponding mapper and reducer functions, respectively.
    ## Provided that 'scams' dictionary is required for both jobs, it is essential to use this function twice.
    ## Failing to do so would result in accessing a potentially empty dictionary,
    ## given the dictionary 'scams' may be loaded on a separate (virtual) machine, given the MR architecture.
```

```
def addr_filter(self):

    with open("scams.json") as f:
        scams_data = jsn.load(f)

        for k, v in scams_data['result'].items():
            self.scams[k] = {
                'category': v.get('category'),
                'subcategory': v.get('subcategory'),
                'url': v.get('url'),
                'status': v.get('status')
                }

## Mapping transaction addresses
## (the to_address field, assuming from_address would be that of the victim being scammed),
## which have a match in the 'scams' dictionary loaded previously,
## with the transaction value the victim has been targeted for.
def scam_addr_mapper(self, _, line):
    fields = line.split(",")
    try:
        if len(fields) == 7:
            scam_addr = fields[2]
            transaction_value = int(fields[3])

            if scam_addr in self.scams:
                yield scam_addr, transaction_value
    except:
        pass

## Similar as in previous discussions, aggregating the scam address locally (i.e., for a mapper), and
## yielding the sum of all its corresponding transaction values (i.e., the Ether/Wei victims have sent).
def scam_addr_combiner(self, scam, values):
    yield scam, sum(values)

## Similar as in previous discussions,
## the tuples are sent to 1 separate reducer in a secondary job.
## Constant key is "lucrative" on this occasion.
def scam_addr_reducer(self, scam, values):
    yield "lucrative", (scam, sum(values))

## The single (executing) reducer finds the scam addres
## whose corresponding value in tuple (scam address, total value of transactions)
## is the maximum among all keys (i.e., among all scam addresses)
## in the tuple (i.e., bundle) 'scam_data' received from the first MR job.
## Once found, the scam address details are pulled from the 'scams' dictionary the 'reducer_init' has (re)loaded,
## which includes category, subcategory, url & status.
def get_lucrative_scam(self, _, scam_data):
    most_lucrative_scam = max(scam_data, key=lambda x: x[1])

    scam_addr = str(most_lucrative_scam[0])
    scam_total = most_lucrative_scam[1]
    scam_category = self.scams.get(scam_addr).get('category')
    scam_subcategory = self.scams.get(scam_addr).get('subcategory')
    scam_url = self.scams.get(scam_addr).get('url')
    scam_status = self.scams.get(scam_addr).get('status')

    result  = f"Most lucrative scam is {scam_addr}" + "\n"
    result += f"Totalling {scam_total} in sent Wei" + "\n"
    result += f"Belonging to \"{scam_category}\" category, and subcategory \"{scam_subcategory}\"" + "\n"
    result += f"With URL: \"{scam_url}\"" + "\n"
    result += f"Being \"{scam_status}\" when the dataset has been compiled" + "\n"

    print(result)

    yield "JOB", "SUCCESSFUL"

def steps(self):
    return [MRStep(mapper_init=self.addr_filter,
                   mapper=self.scam_addr_mapper, combiner=self.scam_addr_combiner, reducer=self.scam_addr_reducer),
            MRStep(reducer_init = self.addr_filter, reducer=self.get_lucrative_scam)]


if __name__ == '__main__':
    ETH_LucativeScams.run()
```

Final Output for *Most Lucrative Scams*:

```
"JOB"   "SUCCESSFUL"
Most lucrative scam is 0x311f71389e3de68f7b2097ad02c6ad7b2dde4c71
Totalling 16709083588073530571339 in sent Wei
Belonging to "Scamming" category, and subcategory "Ponzi"
With URL: "http://333eth.io"
Being "Active" when the dataset has been compiled


As it turns out, http://333eth.io is still in operation, according to the WHOIS history:

"createdDateRaw": "2018-08-27 06:21:04 UTC",
"updatedDateRaw": "2019-08-25 15:14:42 UTC",
"expiresDateRaw": "2021-08-27 06:21:04 UTC"

... provided by https://whois-history.whoisxmlapi.com/
```

MACHINE LEARNING: Price Forecasting

```
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1608030573468_2106
```

In [ ]:

```
from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import unix_timestamp, monotonically_increasing_id, col
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from datetime import datetime

sc = SparkContext()
spark = SparkSession(sc)


## Importing the dataset for train & test from CoinDesk...
def data_import(file = ""):
    if file == "":
        return None

    df = spark.read.format("csv").load(file, inferSchema=True, header=True)

    print("\n\n\nImported Train/Test Dataset Preview:")
    df.show(5)

    return df
```

```
def data_preprocessing(df):

    ## Discarding unnecessary column 'Currency'
    df = df.drop('Currency')

    ## While 'Date' is not considered a feature provided it is an object,
    ## 'Closing Price (USD)' is the label.
    ## Thus, the rest of the columns after discarding the two would represent the feature columns
    feature_cols = df.drop('Date', 'Closing Price (USD)').columns

    ## 'Date' is already formatted, thus casting to 'timestamp' without converting.
    ## Although not a feature column, it is kept in the DataFrame for plotting the Series later.
    df = df.withColumn('Date', df.Date.cast('timestamp'))

    return df, feature_cols


## Assembling the features vector to be used by the Linear Regressor later.
## This will add a new column called "features" (as named below) containing
## the values, per row, from each nominated feature column.
def feature_vector(feature_columns):

    assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
    assembler = assembler.transform(data)

    return assembler


def lregresor(train, test):

    ## The Linear Regressor will use the newly added column "features" (as nominated below).
    ## It will then make predictions based on the "Closing Price (USD)" labeled training set.
    lr = LinearRegression(featuresCol="features", labelCol="Closing Price (USD)")

    ## Training on the labeled dataset...
    model = lr.fit(train)

    ## Evaluating on the lable-free test dataset
    evaluation = model.evaluate(test)
    print("Absolute mean error: ", evaluation.meanAbsoluteError)
    print("Mean squared error, root: ", evaluation.rootMeanSquaredError)
    print("r2: ", evaluation.r2)

    ## Predicting on the lable-free test dataset
    predict = model.transform(test)
    return predict, evaluation


## Computing a datetime object from a given date string.
def get_timeline(str_date):
    return datetime.strptime(str_date, "%Y-%m-%d")


## To plot the predictions and the true values, it is essential to export to CSV
## For plotting a time Series it is also required to sort the dataset, before exporting.
def fields_to_csv(df, fields=None, timeline=False, timeline_col=None, file="output"):

    if fields is None:
        return None

    sel = df.select(fields)

    if timeline is True:
        if timeline_col is None:
            return None
        sel = sel.sort(timeline_col, ascending=True)

    sel = sel.persist()

    print("\n\n\nSaved " + file + "Dataset Preview:")
    sel.show(n=20, truncate=False)

    sel.write.save(file + ".csv", format="csv")


data = data_import(file="eth_USD_daily_prices.csv")

data, feature_columns = data_preprocessing(df=data)

assembled_data = feature_vector(feature_columns=feature_columns)

## The train dataset is until 2019-June
train = assembled_data[assembled_data['Date'] < get_timeline("2019-06-01")]
print("\n\n\nTrain Dataset Preview, before Just 1st, 2019:")
train.show(5)

## Whereas the test dataset is after 2019-June, inclusive
test = assembled_data[assembled_data['Date'] >= get_timeline("2019-06-01")]
print("\n\n\nTest Dataset Preview, after June 1st, 2019 (inclusive):")
test.show(5)

predictions, _ = lregresor(train=train, test=test)

## Because the Linear Regressor creates (which it returns) a new ("predict") DataFrame,
## the 'Date' column dtype is changed to bigint, which requires to be casted back
## to 'timestamp' for conversion and format.
predictions = predictions.withColumn('Date', predictions.Date.cast('timestamp'))

print("Predicted values: ")
predictions.show(5)

## Displaying the final dtypes to ensure the 'Date' is casted back to 'timestamp'
print(predictions.dtypes)

## It is sufficient to export the columns nominated below in order to plot by 'Date':
## the true 'Closing Price (USD)' and its predicted values.
## Thus, the deviations will be visible by plotting the two columns in 2 different colours.
fields_to_csv(df=predictions, fields=['Date', 'Closing Price (USD)', 'prediction'],
              timeline=True, timeline_col='Date',
              file="EVAL_eth_USD_forecasting")
```

Final Output for *Price Forecasting*:

```
Imported Train/Test Dataset Preview:
+--------+-------------------+-------------------+--------------+--------------+-------------+
|Currency|               Date|Closing Price (USD)|24h Open (USD)|24h High (USD)|24h Low (USD)|
+--------+-------------------+-------------------+--------------+--------------+-------------+
|     ETH|2015-08-09 00:00:00|           0.909046|      1.749289|       1.91654|     0.794497|
|     ETH|2015-08-10 00:00:00|           0.692321|      0.909046|      0.909046|     0.692321|
|     ETH|2015-08-11 00:00:00|           0.668067|      0.692321|      0.692321|     0.654331|
```

```
|     ETH|2015-08-12 00:00:00|           0.850151|      0.668067|      1.148621|     0.668067|
|     ETH|2015-08-13 00:00:00|           1.266023|      0.850151|      1.266023|     0.850151|
+--------+-------------------+-------------------+--------------+--------------+-------------+
only showing top 5 rows



Train Dataset Preview, before Just 1st, 2019:
+-------------------+-------------------+--------------+--------------+-------------+--------------------+
|               Date|Closing Price (USD)|24h Open (USD)|24h High (USD)|24h Low (USD)|            features|
+-------------------+-------------------+--------------+--------------+-------------+--------------------+
|2015-08-09 00:00:00|           0.909046|      1.749289|       1.91654|     0.794497|[1.749289,1.91654...|
|2015-08-10 00:00:00|           0.692321|      0.909046|      0.909046|     0.692321|[0.909046,0.90904...|
|2015-08-11 00:00:00|           0.668067|      0.692321|      0.692321|     0.654331|[0.692321,0.69232...|
|2015-08-12 00:00:00|           0.850151|      0.668067|      1.148621|     0.668067|[0.668067,1.14862...|
|2015-08-13 00:00:00|           1.266023|      0.850151|      1.266023|     0.850151|[0.850151,1.26602...|
+-------------------+-------------------+--------------+--------------+-------------+--------------------+
only showing top 5 rows



Test Dataset Preview, after June 1st, 2019 (inclusive):
+-------------------+-------------------+--------------+--------------+--------------+--------------------+
|               Date|Closing Price (USD)|24h Open (USD)|24h High (USD)| 24h Low (USD)|            features|
+-------------------+-------------------+--------------+--------------+--------------+--------------------+
|2019-06-01 00:00:00|     262.6178288677|253.2541136244|264.0152553841|208.6338413696|[253.2541136244,2...|
|2019-06-02 00:00:00|     268.0068681369|262.6332284343|275.0262938829|261.3622701767|[262.6332284343,2...|
|2019-06-03 00:00:00|     269.2745679847|268.0440842728|273.8313452144|210.3151090555|[268.0440842728,2...|
|2019-06-04 00:00:00|     257.9038145283|269.3504439897| 270.903206546|211.3894775572|[269.3504439897,2...|
|2019-06-05 00:00:00|     239.0754424541|249.6732556296|250.8096299108|233.7830046692|[249.6732556296,2...|
+-------------------+-------------------+--------------+--------------+--------------+--------------------+
only showing top 5 rows

('Absolute mean error: ', 3.978951444404736)
('Mean squared error, root: ', 8.393219877712532)
('r2: ', 0.9940054885999584)

Predicted values:
+--------------+-------------+--------------+--------------+--------------+--------------------+------------------+
|Date          |Closin.. (USD)|24h Open(USD) | 24h High(USD)| 24h Low(USD) |            features|    prediction    |
+--------------+-------------+--------------+--------------+--------------+--------------------+------------------+
|2019-06-01 ...|262.6178288677|253.2541136244|264.0152553841|208.6338413696|[253.2541136244,2...| 232.0999236411599|
|2019-06-02 ...|268.0068681369|262.6332284343|275.0262938829|261.3622701767|[262.6332284343,2...| 271.1591489728893|
|2019-06-03 ...|269.2745679847|268.0440842728|273.8313452144|210.3151090555|[268.0440842728,2...|233.79506908245597|
|2019-06-04 ...|257.9038145283|269.3504439897| 270.903206546|211.3894775572|[269.3504439897,2...|231.14512921000338|
|2019-06-05 ...|239.0754424541|249.6732556296|250.8096299108|233.7830046692|[249.6732556296,2...|238.64487730130952|
+--------------+-------------+--------------+--------------+--------------+--------------------+------------------+
only showing top 5 rows

[('Date', 'timestamp'), ('Closing Price (USD)', 'double'), ('24h Open (USD)', 'double'), ('24h High (USD)', 'double'), ('24h Low (USD)', 'double'),
('features', 'vector'), ('prediction', 'double')]



Saved EVAL_eth_USD_forecastingDataset Preview:
+-------------------+-------------------+------------------+
|Date               |Closing Price (USD)|prediction        |
+-------------------+-------------------+------------------+
|2019-06-01 00:00:00|262.6178288677     |232.0999236411599 |
|2019-06-02 00:00:00|268.0068681369     |271.1591489728893 |
|2019-06-03 00:00:00|269.2745679847     |233.79506908245597|
|2019-06-04 00:00:00|257.9038145283     |231.14512921000338|
|2019-06-05 00:00:00|239.0754424541     |238.64487730130952|
|2019-06-06 00:00:00|246.3609227896     |224.67498693104832|
|2019-06-07 00:00:00|243.5358114445     |216.84140114673988|
|2019-06-08 00:00:00|250.9237807026     |204.75158154670964|
|2019-06-09 00:00:00|245.1594274535     |209.67211690538184|
|2019-06-10 00:00:00|229.507045803      |204.68263988261623|
|2019-06-11 00:00:00|243.6975775364     |241.9732265111242 |
|2019-06-12 00:00:00|245.6023534252     |196.71117030247564|
|2019-06-13 00:00:00|257.0824435563     |219.3253054424364 |
|2019-06-14 00:00:00|255.8128945716     |217.8714762097237 |
|2019-06-15 00:00:00|263.9902061782     |211.25771422078242|
|2019-06-16 00:00:00|268.8329563868     |222.89188046091516|
|2019-06-17 00:00:00|270.7520786339     |236.31312833485475|
|2019-06-18 00:00:00|273.8801723339     |270.66274372044245|
|2019-06-19 00:00:00|263.1083721665     |264.20427112363876|
|2019-06-20 00:00:00|268.636551063      |216.9105073678109 |
+-------------------+-------------------+------------------+
only showing top 20 rows
```

FURTHER ANALYSIS

1. Fork the Chain

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1608030573468_2077

In [ ]:

```
import functools
from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.sql.functions import col
from datetime import datetime, timedelta

sc = SparkContext()
spark = SparkSession(sc)


## Computing the datetime 2 weeks before a given date
def timeline_drilldown(from_date, weeks):
    return datetime.strftime(datetime.strptime(from_date, "%Y-%m-%d") - timedelta(weeks=weeks), "%Y-%m-%d")


## Computing the datetime 2 weeks after a given date
def timeline_drillup(from_date, weeks):
    return datetime.strftime(datetime.strptime(from_date, "%Y-%m-%d") + timedelta(weeks=weeks), "%Y-%m-%d")


## Union operation to merge the records dated before and after the fork_timing date
def union_filtered_records(df):
    return functools.reduce(lambda before, after: before.union(after.select(after.columns)), df)


def fork_analysis(records, fork_timing, weeks):
    before_fork_timing = timeline_drilldown(fork_timing, weeks)

    ## Filtering out transactions made more than a specified number of weeks BEFORE the 'fork_timing'
    ## Filtering out transactions with value=0
    before_fork_records = \
        records[(before_fork_timing < records['block_timestamp']) & (records['block_timestamp'] < fork_timing) &
                (records['value'] > 0)]

    ## Converting from Wei to Ether
    before_fork_records = before_fork_records.withColumn('value', before_fork_records.value / (10 ** 18))
    before_fork_records = before_fork_records.persist()

    print("\n\n\nRecords " + str(weeks) + " weeks before fork timing " + fork_timing)
    before_fork_records.show(5, False)

    after_fork_timing = timeline_drillup(fork_timing, weeks)

    ## Filtering out transactions made more than a specified number of weeks AFTER the 'fork_timing'
    ## Filtering out transactions with value=0
    after_fork_records = \
        records[(fork_timing < records['block_timestamp']) & (records['block_timestamp'] < after_fork_timing) &
                (records['value'] > 0)]

    ## Converting from Wei to Ether
    after_fork_records = after_fork_records.withColumn('value', after_fork_records.value / (10 ** 18))
    after_fork_records = after_fork_records.persist()

    print("\n\nRecords " + str(weeks) + " weeks after fork timing " + fork_timing)
    after_fork_records.show(5, False)

    ## Appending the records of both sets (before and after fork_timing date) using a union operation
    analysis_results = union_filtered_records([before_fork_records, after_fork_records])

    ## Sorting in ascending order the resulting dataset
    analysis_results = analysis_results.sort('block_timestamp', ascending=True)
    analysis_results = analysis_results.persist()
    print("\n\n" + str(analysis_results.count()) + " records found " + str(weeks)
          + " weeks before and after fork timing " + fork_timing)
    analysis_results.show(10, False)

    return analysis_results
```

```
## Accounting for smaller yet frequent transactions made from and towards the same addresses on multiple occasions,
## it is neccessary to aggregate the "to_address", "from_address" and "block_timestamp" fields
## by the sum of their transactions.
def get_top_fork_exploiters(df=None, top=10, weeks=2, fork_timing=None):

    if df is None:
        return None

    top_exploiters = df.sort('value', ascending=False)
    top_exploiters = top_exploiters.select(col('block_timestamp').cast("date"), 'from_address', 'to_address', 'value').\
        persist()
    print("\n\nTop" + str(top) + " \"Most Valuable Transactions\" made within" + str(weeks)
          + " weeks before and after fork timing " + fork_timing)
    top_exploiters.show(top, False)

    print("\n\nTop" + str(top) + " \"Most Valuable Transactions\" made within" + str(weeks)
          + " weeks before and after fork timing " + fork_timing
          + ", aggregated by date, senders and receivers")
    top_exploiters.groupby('block_timestamp', 'from_address', 'to_address').\
        sum().sort('sum(value)', ascending=False).show(top, False)

    print("\n\nTop" + str(top) + " \"Highest Sellers\" by total value of transactions made within" + str(weeks)
          + " weeks before and after fork timing " + fork_timing
          + ", aggregated by date and address")
    top_exploiters.groupby('block_timestamp', 'from_address')\
        .sum().sort('sum(value)', ascending=False).show(top, False)

    print("\n\nTop" + str(top) + " \"Highest Sellers\" by total value of transactions made within" + str(weeks)
          + " weeks before and after fork timing " + fork_timing
          + ", aggregated by address ONLY")
    top_exploiters.groupby('from_address')\
        .sum().sort('sum(value)', ascending=False).show(top, False)

    print("\n\nTop" + str(top) + " \"Highest Buyers\" by total value of transactions made within" + str(weeks)
          + " weeks before and after fork timing " + fork_timing
          + ", aggregated by date and address")
    top_exploiters.groupby('block_timestamp', 'to_address')\
        .sum().sort('sum(value)', ascending=False).show(top, False)

    print("\n\nTop" + str(top) + " \"Highest Buyers\" by total value of transactions made within" + str(weeks)
          + " weeks before and after fork timing " + fork_timing
          + ", aggregated by address ONLY")
    top_exploiters.groupby('to_address')\
        .sum().sort('sum(value)', ascending=False).show(top, False)


## To display the charts accordingly, it is essential to sort and save the sorted dataset to CSV.
def fields_to_csv(df, fields=None, timeline=False, timeline_col=None, file="output"):

    if fields is None:
        return None

    sel = df.select(fields)

    if timeline is True:
        if timeline_col is None:
            return None
        sel = sel.sort(timeline_col, ascending=True)

    sel = sel.persist()

    print("\n\n\nSaved Dataset Preview:")
    sel.show(n=20, truncate=False)

    sel.write.save(file + ".csv", format="csv")


transactions = spark.read.csv('/data/ethereum/transactions/*.csv', header=True, inferSchema=True).persist()
transactions = transactions.withColumn('block_timestamp', transactions.block_timestamp.cast('timestamp')).persist()

constantin_fork_timing = "2019-02-28"

analysis = fork_analysis(transactions, constantin_fork_timing, 2)

fields_to_csv(df=analysis, fields=['block_timestamp', 'value'],
              timeline=True, timeline_col='block_timestamp',
              file="fork-analysis-results")

get_top_fork_exploiters(analysis, top=10, weeks=2, fork_timing=constantin_fork_timing)
```

Final Output for *Fork the Chain*:

```
Records 2 weeks before fork timing 2019-02-28
+------------+----------------------------------------+----------------------------------------+--------------+------+----------+-------------------+
|block_number|from_address                            |to_address                              |value         |gas   |gas_price |block_timestamp    |
+------------+----------------------------------------+----------------------------------------+--------------+------+----------+-------------------+
|7234422     |0x829bd824b016326a401d083b33d092293333a830|0xe7eecc14a651b102260f2f73658fcd35d0b9bf98|5.164885470860|210000|5100000000|2019-02-18 03:11:27|
|7234422     |0x829bd824b016326a401d083b33d092293333a830|0x11e68c5caae037d658e95386763a60bfd5417c9b|5.155975354783|210000|5100000000|2019-02-18 03:11:27|
|7234422     |0x829bd824b016326a401d083b33d092293333a830|0x085e8534788d5415ad64ed5b4c804d63018c76ef|5.125145413904|210000|5100000000|2019-02-18 03:11:27|
|7234422     |0x829bd824b016326a401d083b33d092293333a830|0x8999c89fc7e9dbd54c331b43a09ac01a3446ced8|4.996975709998|210000|5100000000|2019-02-18 03:11:27|
|7234422     |0x829bd824b016326a401d083b33d092293333a830|0xafd2b2a2d1f471e46d3ad5b24bde3c09a26f1256|4.808334227589|210000|5100000000|2019-02-18 03:11:27|
+------------+----------------------------------------+----------------------------------------+--------------+------+----------+-------------------+
only showing top 5 rows


Records 2 weeks after fork timing 2019-02-28
+------------+----------------------------------------+----------------------------------------+--------------+------+----------+-------------------+
|block_number|from_address                            |to_address                              |value         |gas   |gas_price |block_timestamp    |
+------------+----------------------------------------+----------------------------------------+--------------+------+----------+-------------------+
|7319495     |0xd84df1c10d526988a3fec89caf2daef5d4b051e6|0x3812a06cd6ef517dc5f87713001c755255cb75d7|6.427000000000|90000 |2000000000|2019-03-07 02:22:09|
|7319495     |0x377de8fce168af3b045e4ba51e819695f9a058a7|0x06012c8cf97bead5deae237070f9587f8e7a266d|0.008000000000|119977|2000000000|2019-03-07 02:22:09|
|7319495     |0x2a65aca4d5fc5b5c859090a6c34d164135398226|0xf521c176a2fd2a2bc13fc00853a435a63236b0bc|0.101867350000|50000 |1000000000|2019-03-07 02:22:09|
|7319495     |0x2a65aca4d5fc5b5c859090a6c34d164135398226|0x2f75682d3b15ae9cc3afce86026bca0fcf05e7f2|0.101796320000|50000 |1000000000|2019-03-07 02:22:09|
|7319495     |0x2a65aca4d5fc5b5c859090a6c34d164135398226|0x05d39941e896ffecbfdaf20ad69608ea8d60237f|0.101678420000|50000 |1000000000|2019-03-07 02:22:09|
+------------+----------------------------------------+----------------------------------------+--------------+------+----------+-------------------+
only showing top 5 rows


6697465 records found 2 weeks before and after fork timing 2019-02-28
+------------+----------------------------------------+----------------------------------------+---------------+------+-----------+-------------------+
|block_number|from_address                            |to_address                              |value          |gas   |gas_price  |block_timestamp    |
+------------+----------------------------------------+----------------------------------------+---------------+------+-----------+-------------------+
|7217184     |0x86588a0606f91939a4fb4a9eaf79a25c4a49bf10|0x349627ba8b2db20e9f569e5dde327dbf4339a7da|0.007000000000 |100000|60000000000|2019-02-14 00:00:21|
|7217184     |0x86588a0606f91939a4fb4a9eaf79a25c4a49bf10|0xc111eeec64ef73c637ab5e2a4f294922077dae0d|0.007000000000 |100000|60000000000|2019-02-14 00:00:21|
|7217184     |0x86588a0606f91939a4fb4a9eaf79a25c4a49bf10|0x349627ba8b2db20e9f569e5dde327dbf4339a7da|0.007000000000 |100000|60000000000|2019-02-14 00:00:21|
|7217184     |0x86588a0606f91939a4fb4a9eaf79a25c4a49bf10|0xc111eeec64ef73c637ab5e2a4f294922077dae0d|0.007000000000 |100000|60000000000|2019-02-14 00:00:21|
```

```
|7217184     |0xbf5ae73fcae491a4a6a0fe86e4412d28213048b1|0x90653a8e41e15833bd5b76bd70cf26f655b25cae|0.007000000000 |100000|60000000000|2019-02-14 00:00:21|
|7217184     |0x5e032243d507c743b061ef021e2ec7fcc6d3ab89|0xba5d6f97f338d4ed9db7faf4252525d3a9cb957e|36.990000000000|45000 |55000000000|2019-02-14 00:00:21|
|7217184     |0xda0e0f03137512c23033638580adc7de1526dfa9|0x9b3abe00201b37bfca0ecc0699e731bb52adea5e|0.010000000000 |21000 |49000000000|2019-02-14 00:00:21|
|7217184     |0x2a5994b501e6a560e727b6c2de5d856396aadd38|0xf9a6074babfd5f0cef37221192ec098f51338f32|0.101911099000 |100000|45000000000|2019-02-14 00:00:21|
|7217184     |0x1ca43b645886c98d7eb7d27ec16ea59f509cbe1a|0x26352d20e6a05e04a1ecc75d4a43ae9989272621|20.152328535939|21000 |40000000000|2019-02-14 00:00:21|
|7217184     |0x3b0bc51ab9de1e5b7b6e34e5b960285805c41736|0x2cb794935d6786d2c50872c13291cd84f58b2a2d|0.088453080000 |21000 |35000000000|2019-02-14 00:00:21|
+------------+------------------------------------------+------------------------------------------+---------------+------+-----------+-------------------+
only showing top 10 rows



Saved Dataset Preview:
+-------------------+----------------+
|block_timestamp    |value           |
+-------------------+----------------+
|2019-02-14 00:00:21|0.007000000000  |
|2019-02-14 00:00:21|0.007000000000  |
|2019-02-14 00:00:21|0.007000000000  |
|2019-02-14 00:00:21|0.007000000000  |
|2019-02-14 00:00:21|0.007000000000  |
|2019-02-14 00:00:21|36.990000000000 |
|2019-02-14 00:00:21|0.010000000000  |
|2019-02-14 00:00:21|0.101911099000  |
|2019-02-14 00:00:21|20.152328535939 |
|2019-02-14 00:00:21|0.088453080000  |
|2019-02-14 00:00:21|119.406510000000|
|2019-02-14 00:00:21|3.330727000000  |
|2019-02-14 00:00:21|5.868000000000  |
|2019-02-14 00:00:21|0.199475000000  |
|2019-02-14 00:00:21|2.161415860000  |
|2019-02-14 00:00:21|0.049845827278  |
|2019-02-14 00:00:21|0.239607790000  |
|2019-02-14 00:00:21|0.840580000000  |
|2019-02-14 00:00:21|0.271986796180  |
|2019-02-14 00:00:21|0.121134042766  |
+-------------------+----------------+
only showing top 20 rows



Top10 "Most Valuable Transactions" made within 2 weeks before and after fork timing 2019-02-28
+---------------+------------------------------------------+------------------------------------------+------------------+
|block_timestamp|from_address                              |to_address                                |value             |
+---------------+------------------------------------------+------------------------------------------+------------------+
|2019-02-25     |0xbe0eb53f46cd790cd13851d5eff43d12404d33e8|0x4e9ce36e442e55ecd9025b9a6e0d88485d628a67|707111.854000000000|
|2019-02-21     |0x75fddbf107ce88ba66ed7fd8182ee6c3a39e9420|0x8b15eb5ea0a405da4d82f26d9197fad62ef7405a|218675.474197810000|
|2019-02-25     |0xbe0eb53f46cd790cd13851d5eff43d12404d33e8|0x030e37ddd7df1b43db172b23916d523f1599c6cb|200888.000000000000|
|2019-03-08     |0x04f2894d12662f2728d02b74ea10056c11467dba|0x60cd748f838651c003c24dbde33c8885d504ed9d|177932.282000000000|
|2019-03-08     |0x42ada615203749550a51a0678b8e7d5f853c6a03|0x391b303cabf70b80e95d9221fd98f7416a1e7275|140450.964589950000|
|2019-03-11     |0x675a67b5deb3da888262338f17cdf342698e8d95|0x60d0cc2ae15859f69bf74dadb8ae3bd58434976b|136581.000048000000|
|2019-03-03     |0x675a67b5deb3da888262338f17cdf342698e8d95|0x60d0cc2ae15859f69bf74dadb8ae3bd58434976b|134588.000165000000|
|2019-03-11     |0x60d0cc2ae15859f69bf74dadb8ae3bd58434976b|0x07c62a47ebe0fa853bb83375e488896ce71266df|110611.402579101160|
|2019-03-08     |0x01babe8f3182a280d791179b2364c635483b060a|0x6eeee9af0788c04971f5f91f0275b6c5b1c49840|106808.000000000000|
|2019-02-27     |0xfbb1b73c4f0bda4f67dca266ce6ef42f520fbb98|0x5e032243d507c743b061ef021e2ec7fcc6d3ab89|97999.990000000000 |
+---------------+------------------------------------------+------------------------------------------+------------------+
only showing top 10 rows



Top10 "Most Valuable Transactions" made within 2 weeks before and after fork timing 2019-02-28, aggregated by date,
senders and receivers
+---------------+------------------------------------------+------------------------------------------+------------------+
|block_timestamp|from_address                              |to_address                                |sum(value)        |
+---------------+------------------------------------------+------------------------------------------+------------------+
|2019-02-25     |0xbe0eb53f46cd790cd13851d5eff43d12404d33e8|0x4e9ce36e442e55ecd9025b9a6e0d88485d628a67|707111.854000000000|
|2019-02-21     |0x75fddbf107ce88ba66ed7fd8182ee6c3a39e9420|0x8b15eb5ea0a405da4d82f26d9197fad62ef7405a|218676.474197810000|
|2019-02-25     |0xbe0eb53f46cd790cd13851d5eff43d12404d33e8|0x030e37ddd7df1b43db172b23916d523f1599c6cb|200888.000000000000|
|2019-03-08     |0x04f2894d12662f2728d02b74ea10056c11467dba|0x60cd748f838651c003c24dbde33c8885d504ed9d|177932.292000000000|
|2019-03-08     |0x42ada615203749550a51a0678b8e7d5f853c6a03|0x391b303cabf70b80e95d9221fd98f7416a1e7275|140450.964589950000|
|2019-03-07     |0x267be1c1d684f78cb4f6a176c4911b741e4ffdc0|0xd3bc722ebd74d8f4bc1e732e02db0133041d3364|137136.108280000000|
|2019-03-11     |0x675a67b5deb3da888262338f17cdf342698e8d95|0x60d0cc2ae15859f69bf74dadb8ae3bd58434976b|136581.000048000000|
|2019-03-10     |0x65043331a41cc4350030bb4011e2a7a85c90ec11|0x675a67b5deb3da888262338f17cdf342698e8d95|136581.000000000000|
|2019-03-03     |0x675a67b5deb3da888262338f17cdf342698e8d95|0x60d0cc2ae15859f69bf74dadb8ae3bd58434976b|134588.000165000000|
|2019-03-03     |0x65043331a41cc4350030bb4011e2a7a85c90ec11|0x675a67b5deb3da888262338f17cdf342698e8d95|134588.000000000000|
+---------------+------------------------------------------+------------------------------------------+------------------+
only showing top 10 rows



Top10 "Highest Sellers" by total value of transactions made within 2 weeks before and after fork timing 2019-02-28,
aggregated by date and address
+---------------+------------------------------------------+------------------+
|block_timestamp|from_address                              |sum(value)        |
+---------------+------------------------------------------+------------------+
|2019-02-25     |0xbe0eb53f46cd790cd13851d5eff43d12404d33e8|988887.854000000000|
|2019-02-21     |0x75fddbf107ce88ba66ed7fd8182ee6c3a39e9420|218676.474197810000|
|2019-02-27     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|213614.994353218471|
|2019-02-19     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|202303.330415770000|
|2019-02-18     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|195910.676801465031|
|2019-03-07     |0x267be1c1d684f78cb4f6a176c4911b741e4ffdc0|184691.922754429600|
|2019-03-08     |0x04f2894d12662f2728d02b74ea10056c11467dba|177932.292000000000|
|2019-02-24     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|159143.346203480000|
|2019-02-18     |0x876eabf441b2ee5b5b0554fd502a8e0600950cfa|143609.029891400000|
|2019-03-01     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|142589.663079230000|
+---------------+------------------------------------------+------------------+
only showing top 10 rows



Top10 "Highest Sellers" by total value of transactions made within 2 weeks before and after fork timing 2019-02-28,
aggregated by address ONLY
+------------------------------------------+------------------+
|from_address                              |sum(value)        |
+------------------------------------------+------------------+
|0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|2902858.917236253502|
|0x876eabf441b2ee5b5b0554fd502a8e0600950cfa|1264837.190522280000|
|0x6cc5f688a315f3dc28a7781717a9a798a59fda7b|1255231.343000000000|
|0x267be1c1d684f78cb4f6a176c4911b741e4ffdc0|1096175.986062051500|
|0xdf95de30cdff4381b69f9e4fa8dddce31a0128df|1033163.690071420000|
```

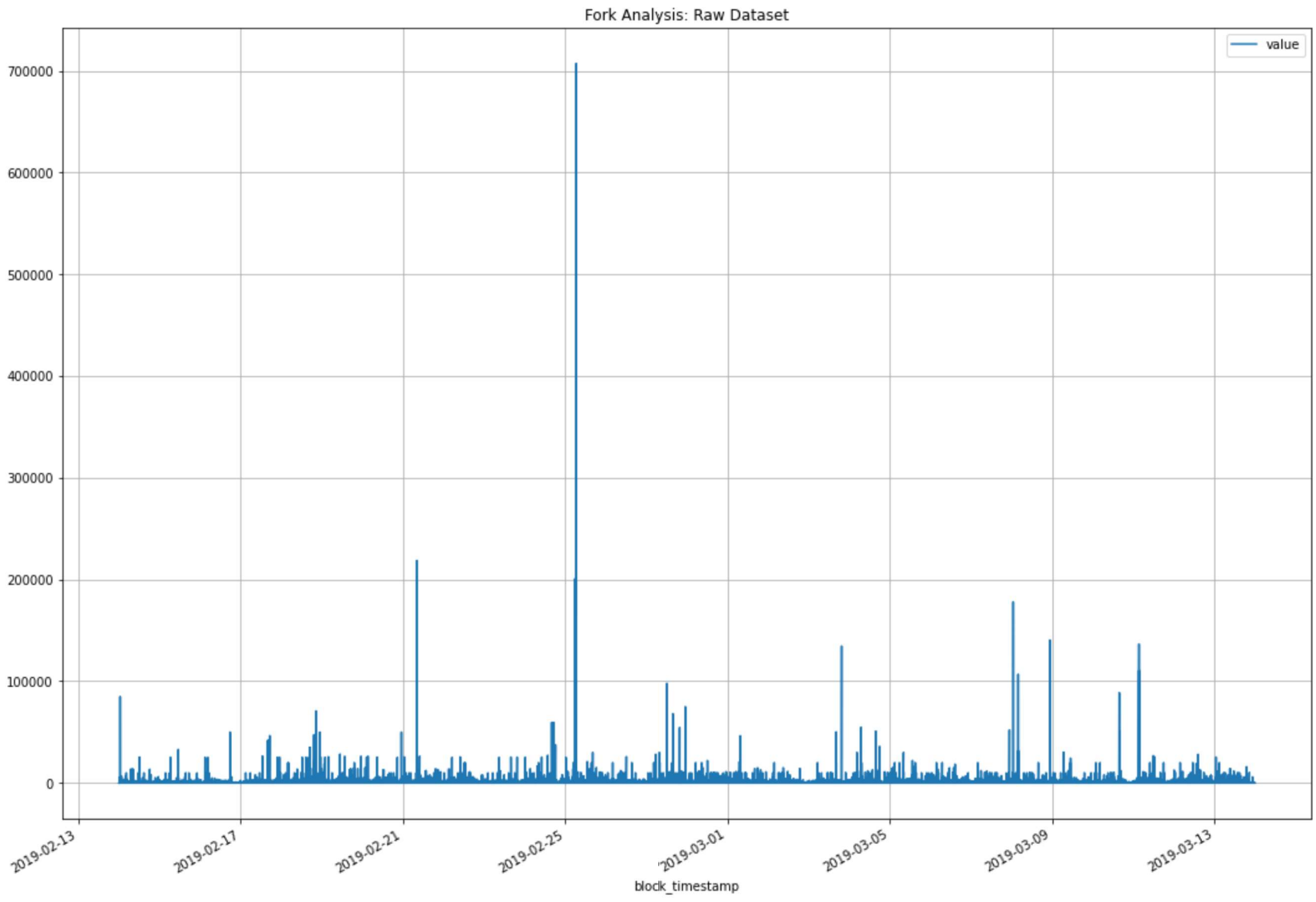
```
|0xb9a4873d8d2c22e56b8574e8605644d08e047549|1021580.719008700000|
|0x6f50c6bff08ec925232937b204b0ae23c488402a|1019361.661835150000|
|0x5e032243d507c743b061ef021e2ec7fcc6d3ab89|999424.639330263415 |
|0xbe0eb53f46cd790cd13851d5eff43d12404d33e8|988887.854000000000 |
|0xf4a2eff88a408ff4c4550148151c33c93442619e|861300.000000000000 |
+----------------------------------------+-------------------+
only showing top 10 rows
```

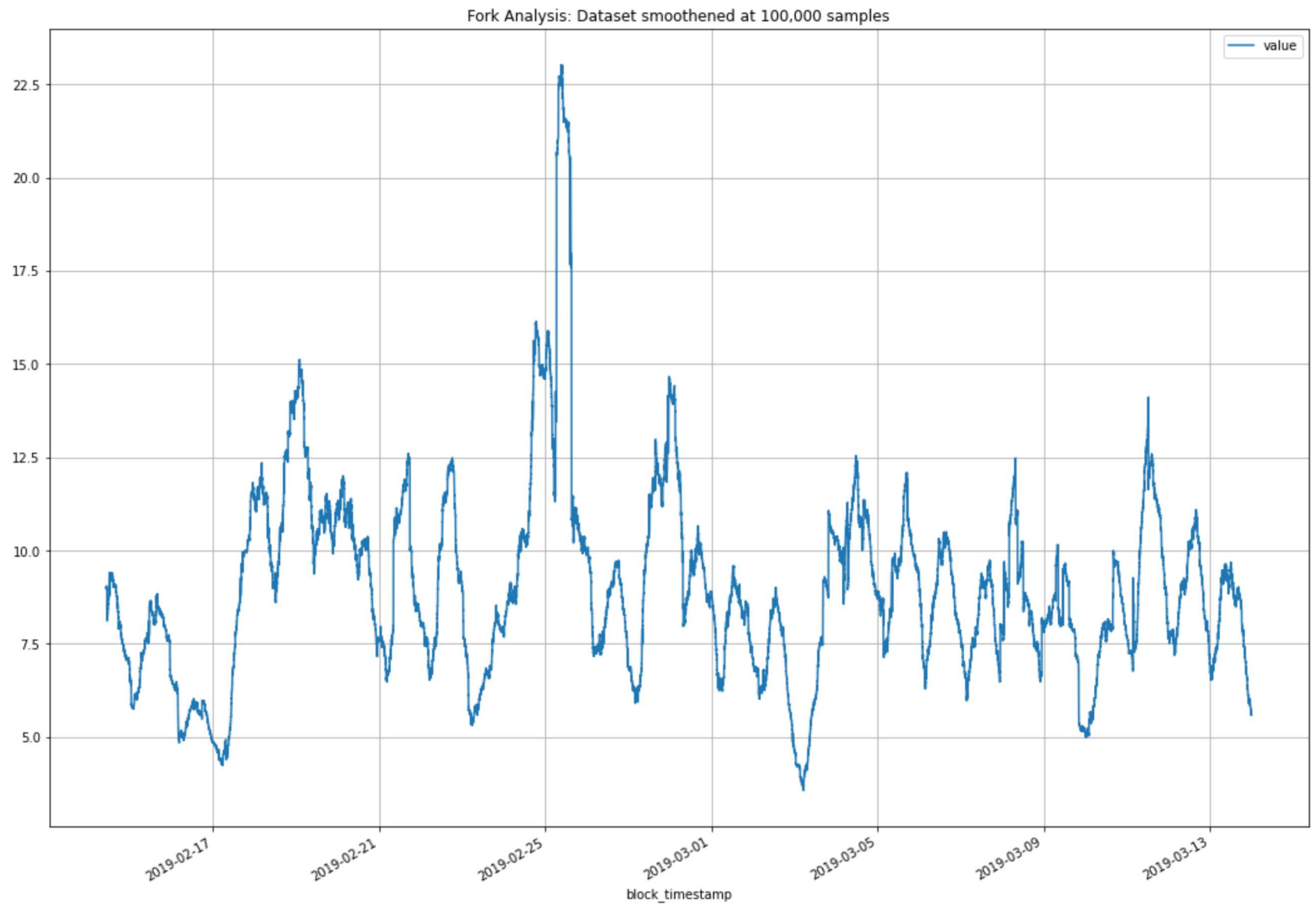
Top10 "Highest Buyers" by total value of transactions made within 2 weeks before and after fork timing 2019-02-28, aggregated by date and address

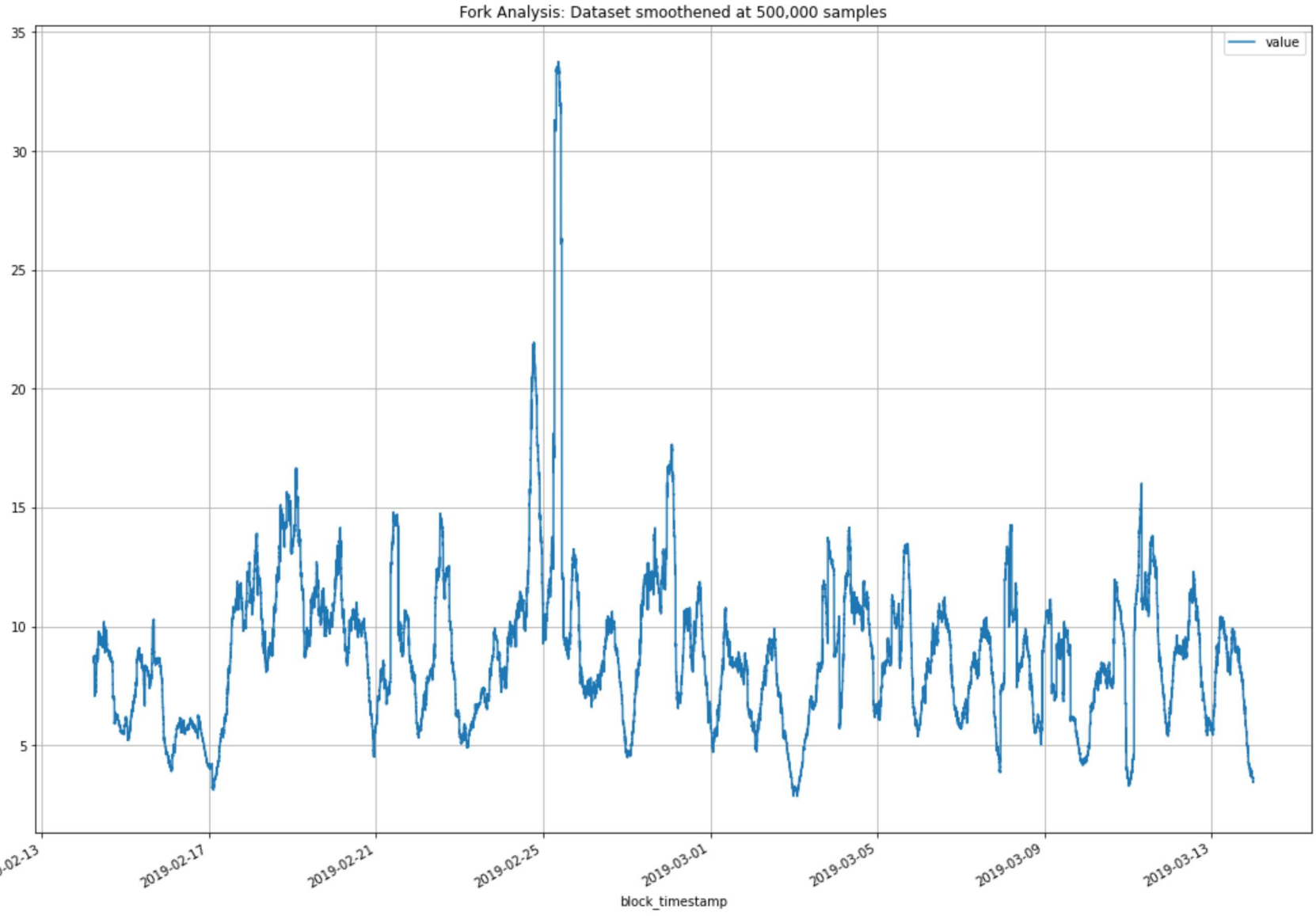
```
+--------------+----------------------------------------+-------------------+
|block_timestamp|to_address                             |sum(value)         |
+--------------+----------------------------------------+-------------------+
|2019-02-25     |0x4e9ce36e442e55ecd9025b9a6e0d88485d628a67|707111.854000000000|
|2019-02-18     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|238430.579661254706|
|2019-02-21     |0x8b15eb5ea0a405da4d82f26d9197fad62ef7405a|218676.474197810000|
|2019-02-27     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|215187.439867497688|
|2019-02-25     |0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|203268.832051277708|
|2019-02-25     |0x030e37ddd7df1b43db172b23916d523f1599c6cb|200888.000000000000|
|2019-03-08     |0x60cd748f838651c003c24dbde33c8885d504ed9d|177932.292000000000|
|2019-03-08     |0x6eeee9af0788c04971f5f91f0275b6c5b1c49840|166433.344000000000|
|2019-03-11     |0x60d0cc2ae15859f69bf74dadb8ae3bd58434976b|154609.922945660558|
|2019-03-09     |0x6cc5f688a315f3dc28a7781717a9a798a59fda7b|153755.635626890000|
+--------------+----------------------------------------+-------------------+
only showing top 10 rows
```

Top10 "Highest Buyers" by total value of transactions made within 2 weeks before and after fork timing 2019-02-28, aggregated by address ONLY

```
+----------------------------------------+-------------------+
|to_address                              |sum(value)         |
+----------------------------------------+-------------------+
|0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be|2877419.183366258706|
|0x876eabf441b2ee5b5b0554fd502a8e0600950cfa|1273667.550318610000|
|0x6cc5f688a315f3dc28a7781717a9a798a59fda7b|1204020.410638420000|
|0x5e032243d507c743b061ef021e2ec7fcc6d3ab89|1053647.095884709118|
|0xdf95de30cdff4381b69f9e4fa8dddce31a0128df|1025697.628859156000|
|0xb9a4873d8d2c22e56b8574e8605644d08e047549|1022329.450221987000|
|0x6f50c6bff08ec925232937b204b0ae23c488402a|1019795.086701436000|
|0xf4a2eff88a408ff4c4550148151c33c93442619e|969655.242000000000 |
|0x4e9ce36e442e55ecd9025b9a6e0d88485d628a67|877109.507523253502 |
|0xfa52274dd61e1643d2205169732f29114bc240b3|876043.108768270746 |
+----------------------------------------+-------------------+
only showing top 10 rows
```







CoinDesk: Ethereum Daily Prices (USD), 17-Feb-2019 to 03-Mar-2019

The peak matches with the transaction made by "3e8", also matching the results from the top10 charts above.

Furthermore, it can be noticed there are many other transactions smaller, but more frequent. Hence, highlighting the importance of exploring frequent transaction of small amounts potentially made by the same entity.
This has been investigated with grouping the transactions by addresses/block_timestamps, which yielded different addresses in terms of results.

The last 2 plots of this task are results of Moving Averages at 100,000 and 500,000 data samples, respectively. This was necessary in order to smoothen the dataset.

Conclusions for *Fork the Chain* analysis:

A close look at "Top10 Highest Buyers" aggregated by date and address, we can see that "0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be" has made several transactions worth various amounts, in the run-up to the time of fork. According to the same top aggregated by address only, this address seems to be the first.
As such, we can declare "0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be" to have taken the most advantage in terms of buying Ether.

Another close look at "Top10 Highest Sellers" aggregated by date and address, we can notice more occurances from the same address, making transactions worth of various amounts in the run-up to the time of work. The same top aggregated by address only also declares "0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be" to be the heighest seller in this period.

Finally, we can declare "0x3f5ce5fbfe3e9af3971dd833d26ba9b5c936f0be" to have taken the most advantage in terms of buying as well as selling Ether in run-up to the "Constantinople" fork. Thus, this individual has taken advantage the most of this fork.

However, we can only say that this "0be" (for short) has only tried to take advantage of the wave, as the figures indicate a -25439.73387 (Ether) trade loss in the highlighted period alone, without taking into account other personal reserves. It is not realistic to claim that this entity with address ending in "0be" has made a fortune.

Although 0xbe0eb53f46cd790cd13851d5eff43d12404d33e8 ("3e8", for short) has been ranked first in other top10 charts, this is only representative for the most valuable transaction, which does not seem to add up to a larger amount (i.e., no other large transactions or smaller but frequent were found). This is an example painting the necessity to group by addresses/block_timestamps, leading to find transactions such as those made by "0be".

According to the "CoinDesk: Ethereum Daily Prices (USD), 17-Feb-2019 to 03-Mar-2019" chart, "0be", who ranks first in both top10 charts, may have pursued the right timing to trade, aiming to take advantage of the rally, but with different amounts and in the wrong context (for instance, buying instead of selling, or vice-versa); If looked at in conjunction with the top10 charts grouped by 'block_timestamp' for both, sellers and buyers.

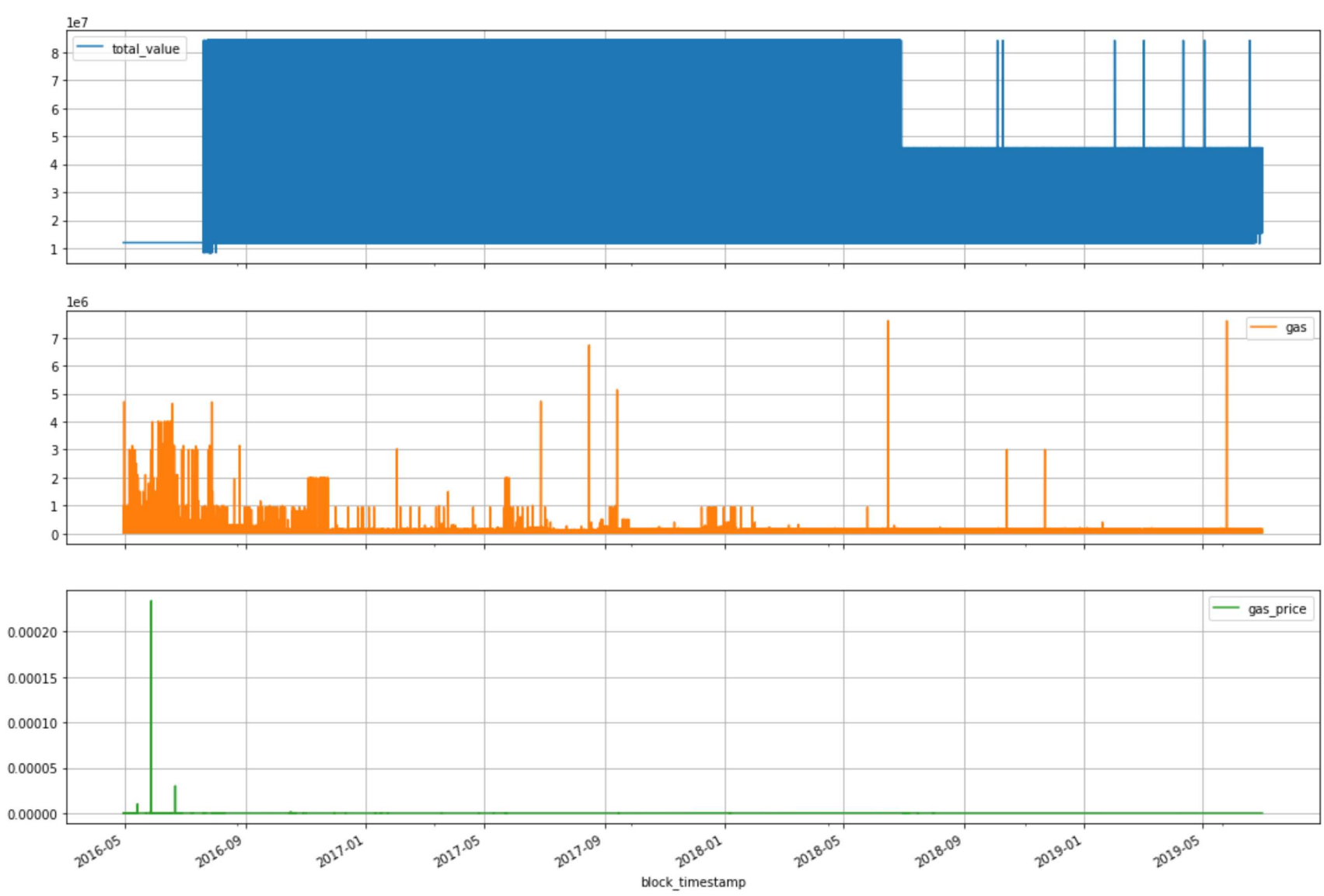
2. Gas Guzzlers

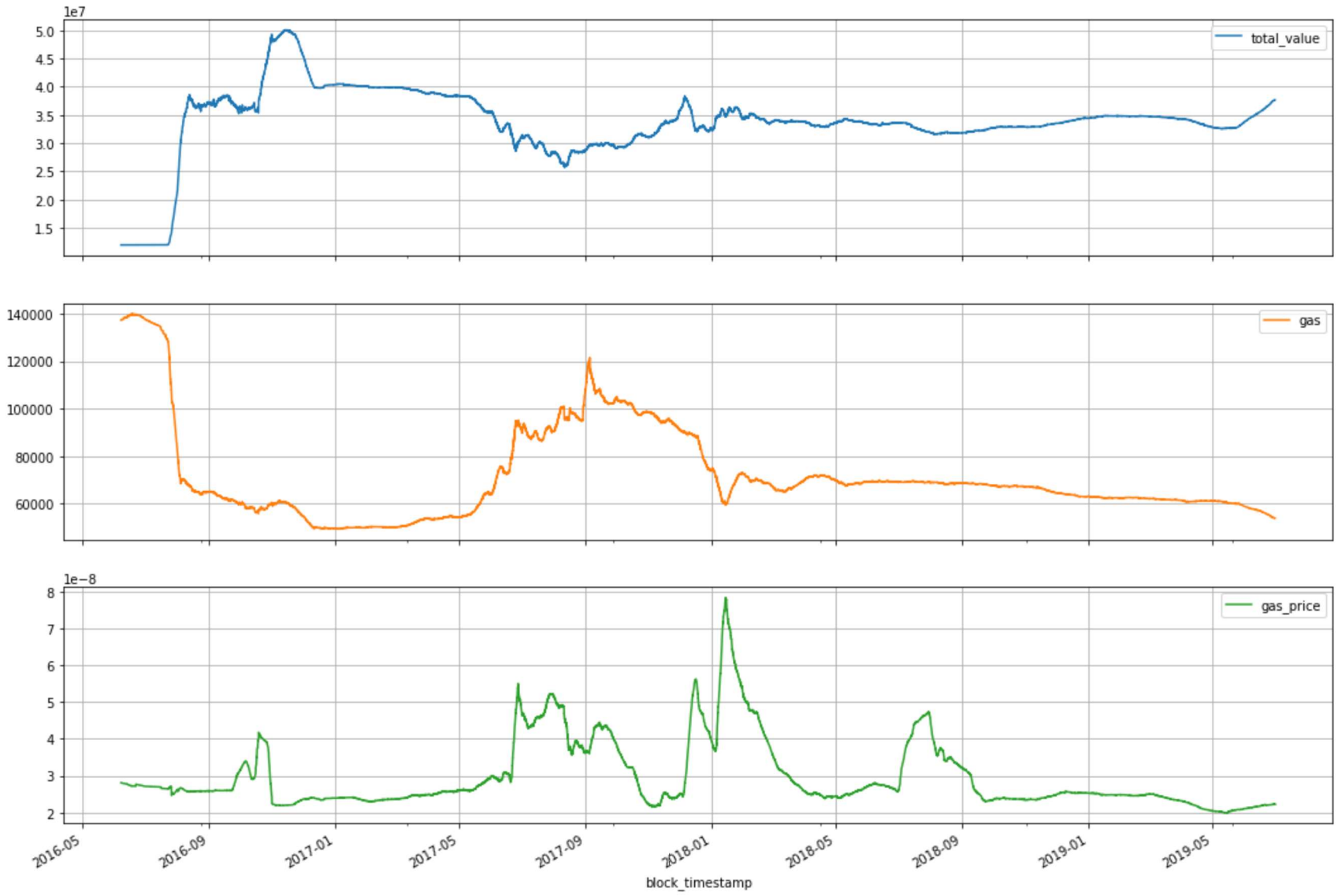
http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1608030573468_2023

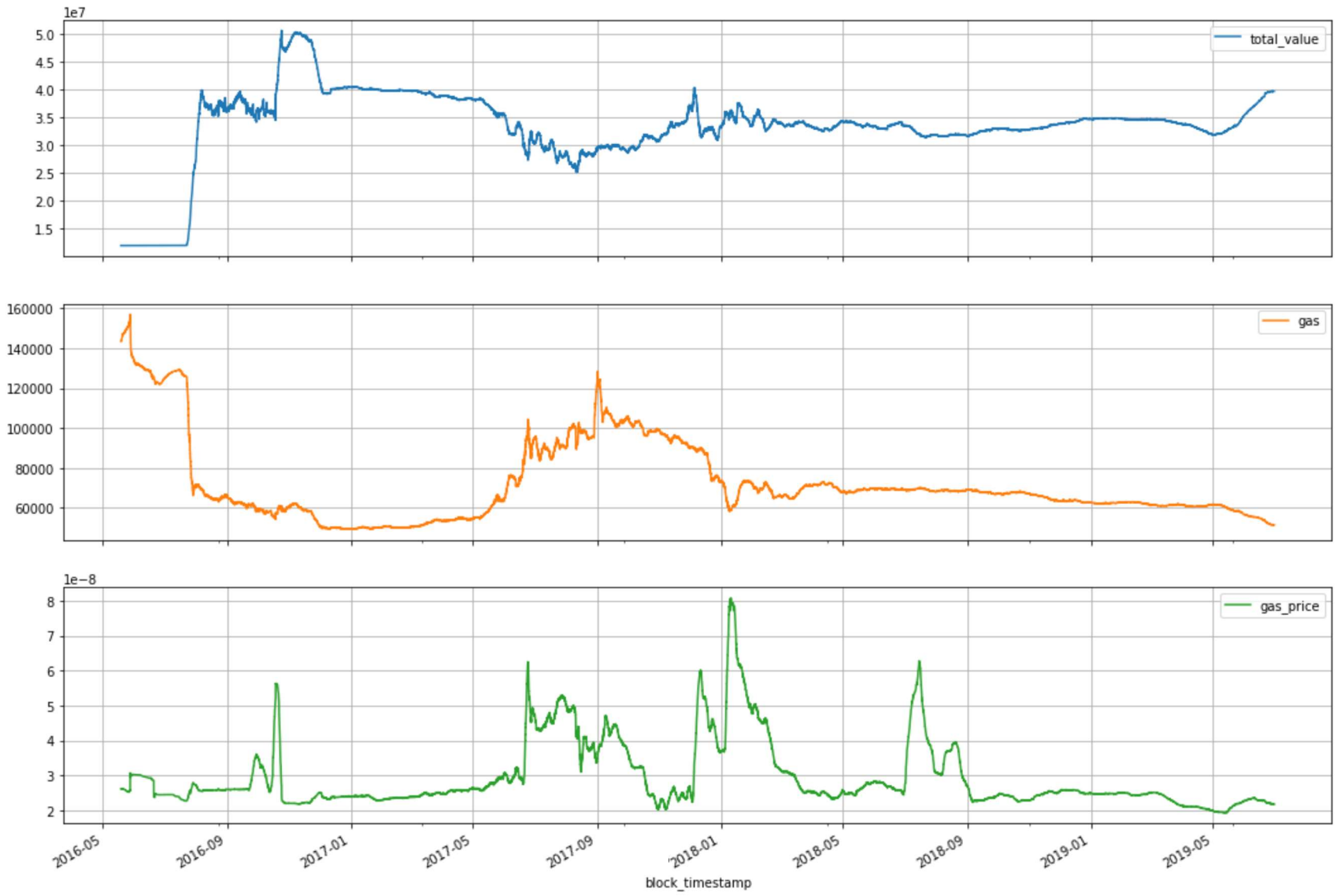
In [ ]:
```
import time
import functools
from datetime import datetime, timedelta
from pyspark import SparkContext
from pyspark.sql import SparkSession, Row
from pyspark.sql.types import *
from pyspark.sql import SparkSession

sc = SparkContext()
spark = SparkSession(sc)


## Importing the top10-services (the final output from Part B)
def data_import(file = ""):
    if file == "":
        return None

    df = spark.read.csv(file, sep=r'\t', inferSchema=True, header=False)

    ## Renaming the columns...
    df = df.withColumnRenamed(existing=df.columns[0], new='contract_addr')
    df = df.withColumnRenamed(existing=df.columns[1], new='total_value')

    ## The final output from Part B comes with the reported values in Wei
    ## Thus, converting from Wei to Ether...
    df = df.withColumn('total_value', df.total_value / (10 ** 18))


    print("\n\n\nTop10 Contracts Dataset Preview:")
    df.show(n=20, truncate=False)

    return df


## Executing an Inner Join on
## to_address from 'Transactions' and 'contract_addr' (renamed) from Part B's final output
```

```python
def joiner(df1=None, df2=None):

    if (df1 is None) or (df2 is None):
        return None

    ## As discussed, the keys on which the Inner Join it is executed on...
    keys = [df1.to_address == df2.contract_addr]
    df = df1.join(df2, on=keys, how='inner')

    ## Discarding unnecessary columns and reordering by selection for increased clarity...
    df = df.select(df.columns[4:])
    df = df.select('contract_addr', 'total_value', 'gas', 'gas_price', 'block_timestamp')

    ## gas_price is also in Wei.
    ## thus, converting from Wei to Ether.
    df = df.withColumn('gas_price', df.gas_price / (10 ** 18))

    print("\n\n\nTransactions Joined Dataset Preview:")
    df.show(n=20, truncate=False)

    return df


## For plotting a time Series, it is essential to sort in ascending order by "block_timestamp".
## Then, exporting for creating the charts.
def fields_to_csv(df, fields=None, timeline=False, timeline_col=None, file="output"):

    if fields is None:
        return None

    sel = df.select(fields)

    if timeline is True:
        if timeline_col is None:
            return None
        sel = sel.sort(timeline_col, ascending=True)

    sel = sel.persist()

    print("\n\n\nSaved Dataset Preview:")
    sel.show(n=20, truncate=False)
    sel.write.save(file + ".csv", format="csv")


contracts = data_import(file="./output-services-top10.tsv")
inmem_c = contracts.persist()

if contracts is None:
    exit(-1)

transactions = spark.read.csv('/data/ethereum/transactions/*.csv', header=True, inferSchema=True)
inmem_t = transactions.persist()

## "block_timestamp" is presented as bigint, thus it needs to be converted to 'timestamp' for formatting...
inmem_t = inmem_t.withColumn('block_timestamp', inmem_t.block_timestamp.cast('timestamp')).persist()

## In order to collect the gas and gas_price values for the contract addresses resulting from Part B,
## an Inner Join must be executed as discussed in the comments above the joiner(..) function.
analysis = joiner(inmem_t, inmem_c)
analysis = analysis.persist()

fields_to_csv(df=analysis, fields=['block_timestamp', 'total_value', 'gas', 'gas_price'],
              timeline=True, timeline_col='block_timestamp',
              file="gas-analysis")
```

Gas Analysis: Raw Dataset







In terms of the gas pricing, there are several steep increases on the smoothened charts, with only a few short periods of stability.
However, it is difficult to draw conclusions from the raw chart, provided the series is far from being smooth (due to the size of the dataset). It is visible, however, a peak closer to 2016-May than 2016-September, with a flattened line in rest.

Gas amount and the total value of transactions seem to be opposing: the value of transactions declining with contracts requiring more gas. For instance, this can be noticed in the smoothened charts (blue and orange) between 2017-May and 2018-January, where the total_value (blue) creating a dent when the gas (orange) amount is increasing.
Besides this correlation, contracts do not appear to have increased the amount of required gas. This has actually been decreasing, from 140,000 units to less than 60,000 units.

The last 2 sets of 3 plots each (of this task) are results of Moving Averages at 100,000 and 500,000 data samples, respectively. This was necessary in order to smoothen the dataset.

3. Comparative Evaluation

http://andromeda.student.eecs.qmul.ac.uk:8088/cluster/app/application_1608030573468_2053

In [ ]:
```
import time
import functools
from datetime import datetime, timedelta
from pyspark import SparkContext
from pyspark.sql import SparkSession, Row
from pyspark.sql.types import *
from pyspark.sql import SparkSession
from operator import add

sc = SparkContext()
spark = SparkSession(sc)

## Executed on the 'Transactions' dataset.
def is_valid_transaction(line):
    fields = line.split(",")

    try:
        ## Dismissing malformed lines (i.e., whose number of fields is not 7 after splitting by ',')
        if len(fields) != 7:
            return False

        ## Sanity check: ensuring field 'value' can be converted to integer
        int(fields[3])
        return True

    except:
        return False


## Executed on the 'Contracts' dataset
def is_valid_contract(line):

    ## Dismissing malformed lines (i.e., whose number of fields is not 5 after splitting by ',')
    return len(line.split(",")) == 5


## Extracting the transaction to_address field and the transaction (as int) value
def addr_value(line):
    fields = line.split(",")

    addr  = fields[2]
    value = int(fields[3])

    return addr, value


## Extracting the contract address from the 'Contracts' dataset.
## No other field is required...
## We are interested in finding matches between these addresses and those from the to_address of 'Transactions' dataset.
## In essence, identifying which addresses in the 'Transactions' dataset from its to_address field are contract addreses.
def contractAddr(line):
    addr = line.split(",")[0]

    return addr, None

### Loading the 'Transactions' dataset
transactions = sc.textFile('/data/ethereum/transactions/*.csv')
### Dismissing malformed lines using filtering
transactions = transactions.filter(is_valid_transaction).persist()

### Mapping to_address to its corresponding transaction value
### Then, aggregating (i.e., reducing) by key (i.e., by to_address) by adding up all transaction values mapped to that key
transactions_agg = transactions.map(addr_value).reduceByKey(add).persist()


### Loading the 'Contracts' dataset
contracts = sc.textFile('/data/ethereum/contracts/*')
### Dismissing malformed lines using filtering
contracts = contracts.filter(is_valid_contract).persist()

### Mapping, without aggregating, in order to collect (distinct) all contract addresses from the 'Contracts' dataset
contracts_addr = contracts.map(contractAddr).persist()

### Executing an inner join for transactions and contracts on keys:
### 'to_address' of 'Transactions' and the distinct contract addresses collected above
joined_tc = transactions_agg.join(contracts_addr).persist()


### The following represents a tuple in joined_tc:
# tup[0]                                        tup[1][0]
# (u'0x5d16d41d6c3466ca319b02ca5d1575dc1497c37b', (441640000000000000, None)),"
### Where, tup[0] is the key (i.e., to_address/contract address) the inner join has been executed on,
### tup[1][0] is the sum of all transaction values (i.e., the contract value) for that key (i.e., the contract address).


### Sorting in desceding order using -tup[1][0], and
### taking only the first 10 records
top10 = joined_tc.takeOrdered(10, key=lambda tup: -tup[1][0])


### Listing the results and wrapping up with a fancy format...
print("Contract Address\t\t\t\tContract Value (in Wei)")
print("----------------\t\t\t\t-------------------------")
for record in top10:
    print(str(record[0]) + "\t" + str(record[1][0]))
```

Conclusions for *Apache Hadoop v. Spark* performance:

```
The total elapsed time for Hadoop/MapReduce:
    Time Elapsed in Job 1 + Time Elapsed in Job 2 + Time Elapsed in Job 3
    <=>
    (2 min 3 seconds) + (56 seconds) + (23 seconds) = 3.36 minutes

The total elapsed time for Spark:
    1 minute, 43 seconds
Spark has outperformed Hadoop/MapReduce by far, being more agile in this task than Apache Hadoop.

Method persist() is an iterative computation support, which Spark features and the implemented algorithm presented
above is using extensively. In essence, Spark is able to reuse RDDs that were created previously by storing these in
the memory; Thus, avoiding to reconstruct these, which otherwise would create a bottleneck should these be used used
again.
```

Final Output for *Comparative Evaluation*: Contract Address v. Total Contract Value (in Wei)

```
Contract Address                            Contract Value (in Wei)
----------------                            -------------------------
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444  84155100809965865822726776
0xfa52274dd61e1643d2205169732f29114bc240b3  45787484483189352986478805
0x7727e5113d1d161373623e5f49fd568b4f543a9e  45620624001350712557268573
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef  43170356092262468919298969
```

```
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8  270689215820195424998828 77
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd  211041951380936600500000 00
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3  155623989568021122547194 09
0xbb9bc244d798123fde783fcc1c72d3bb8c189413  119836087292028938468186 81
0xabbb6bebfa05aa13e908eaa492bd7a8343760477  117064571779408955217704 04
0x341e790174e3a4d35b65fdc067b6b5634a61caea  837900075191775562405750 0
```