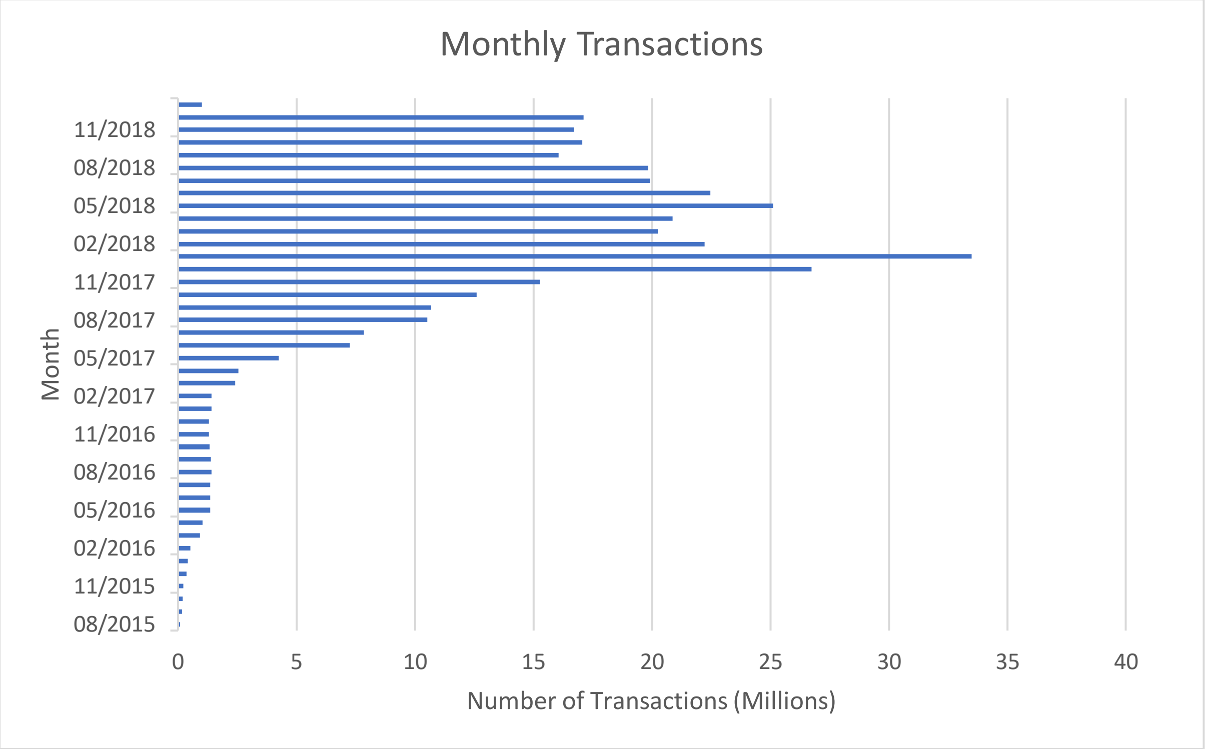
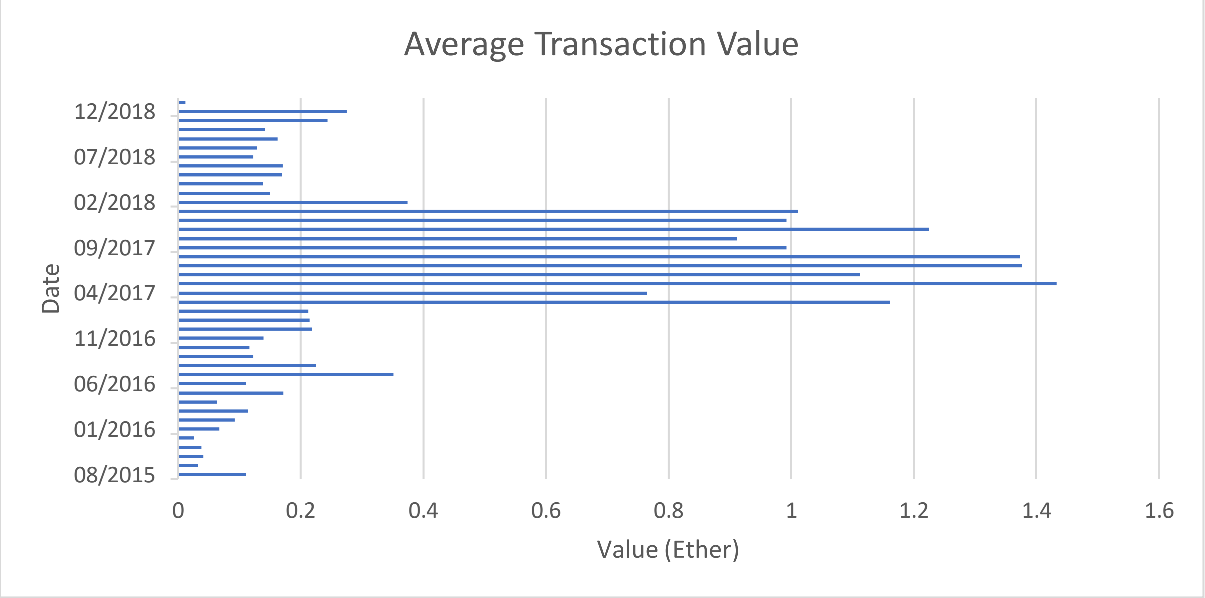
This is an analysis of an Ethereum Dataset from the years 2015 to 2019, the following code used the dataset which was stored in an Amazon S3 bucket

Question A) Time Analysis

The **number of monthly transactions** that occurred between August 2015 and January 2019. The peak month of transaction happened during January 2018.



The **average value of a transaction (in Ether)** between August 2018 and January 2019. The highest average transaction value was 1.43 Ether during May 2017



Question A) Code for Monthly Occurrence of transaction and Average transaction value

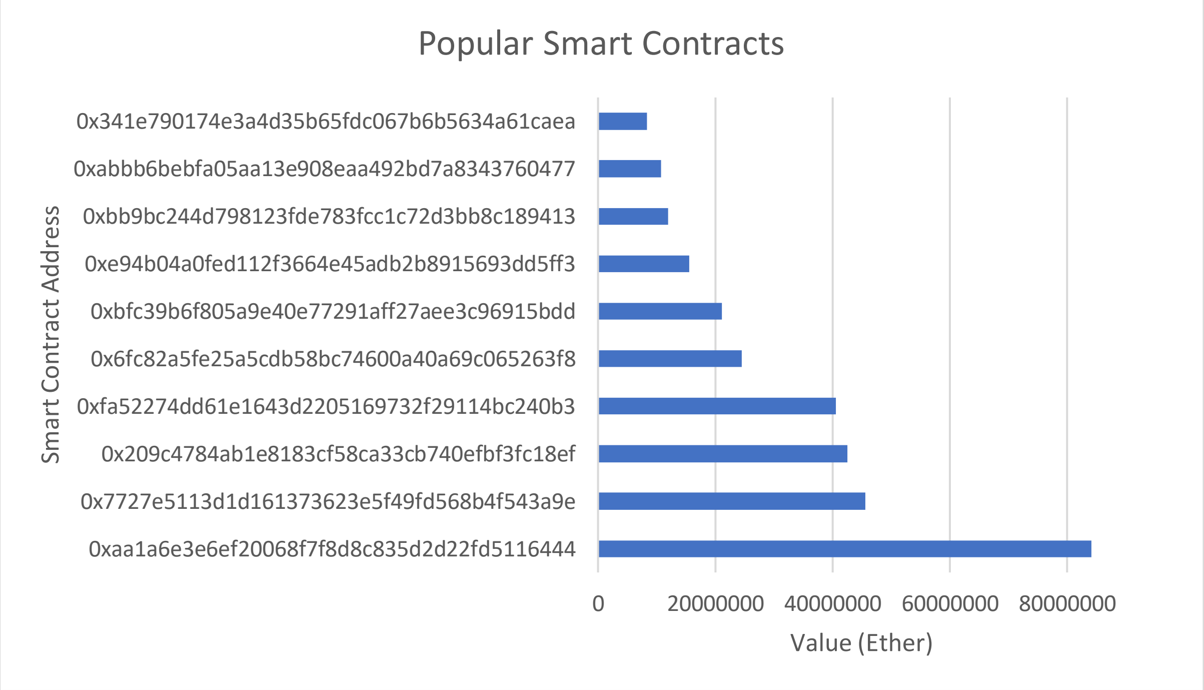
```
import sys, string
import os
import socket
import time
import operator
import boto3
import json
import math
import time
import operator
from pyspark.sql import SparkSession
from datetime import datetime

if __name__ == "__main__":

    spark = SparkSession\
        .builder\
        .appName("Ethereum")\
        .getOrCreate()

    # shared read-only object bucket containing datasets
    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url =
os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
    s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
    s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
    s3_bucket = os.environ['BUCKET_NAME']

    hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
    hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
    hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
    hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
    hadoopConf.set("fs.s3a.path.style.access", "true")
    hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")

    lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket +
"/ethereum-parvulus/transactions.csv")
    header = lines.first()
    clean_sample = lines.filter(lambda x: x != header)
    # Monthly Transactions
    monthly = clean_sample.map(lambda x:
(datetime.fromtimestamp(int(x.split(",")[11])).strftime("%m/%Y"), 1))
    monthly = monthly.reduceByKey(operator.add)
    # Average Transactions
    average = clean_sample.map(lambda x:
(datetime.fromtimestamp(int(x.split(",")[11])).strftime("%m/%Y"),
int(x.split(",")[7])))
```

```
    record = average.count()
    average = average.reduceByKey(operator.add)
    average = average.map(lambda x: (x[0], format(x[1]/record, '.2f')))
```

Firstly, the dataset is filter out such that the header is removed from the RDD. Once the data is filtered out, the mapping function is used to map each timestamp value (which has been converted to the format MM/YYYY) to the value 1. The reduceByKey function groups all values by the timestamp and adds up the value, this would then give the number of transactions that happened for all the months in the dataset. To calculate the average value of a transaction, the mapping function is used to map the timestamp (converted to MM/YYYY) with the Wei transaction value. The count of the number of transactions is recorded and the reduceByKey function reduces and groups the values together such that for each month, there is the total transaction value. Another mapping function is then used to divide each total transaction value by the count of the number of transactions. This would give the average transaction value for each month.

<u>Question B) Top Ten Most Popular Services</u>

The top ten most popular smart contracts where the most popular smart contract received 84155363.7 Ethers (Wei is converted to Ether by dividing by 10^{18}).



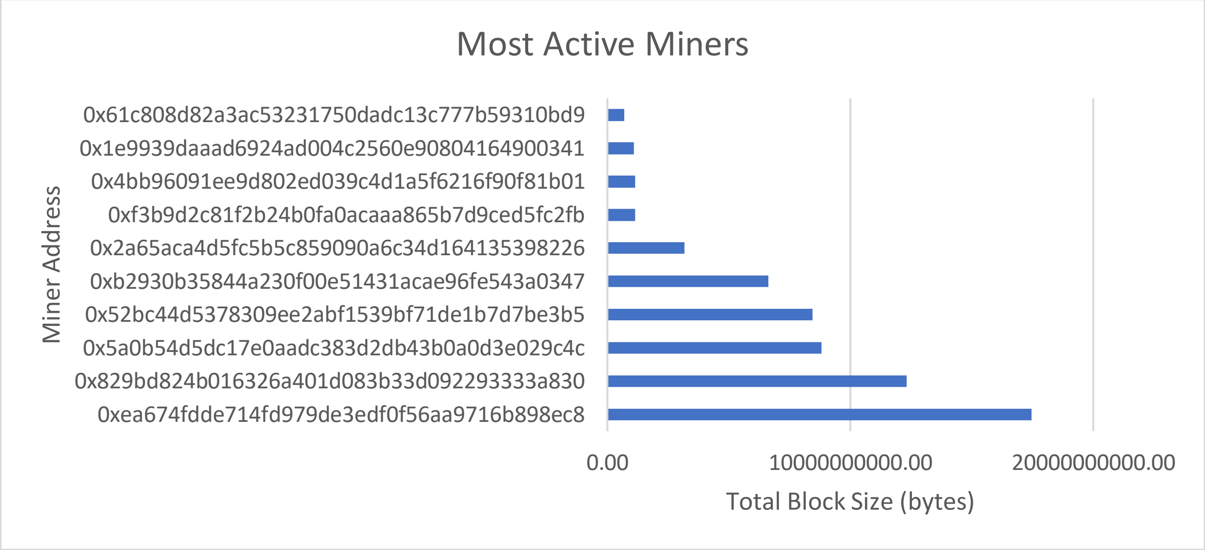
<u>Question B) Code for Top Ten Most Popular Services</u>

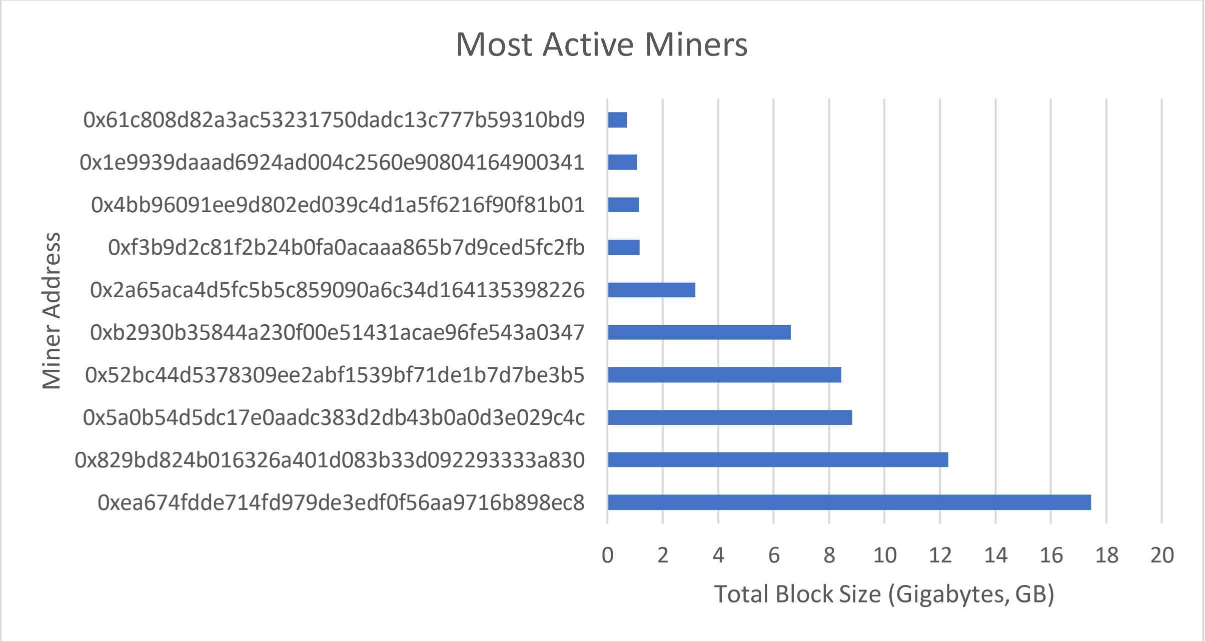
```
import sys, string
import os
import socket
import time
import operator
import boto3
import json
import math
import time
import operator
from pyspark.sql import SparkSession
from datetime import datetime

if __name__ == "__main__":

    spark = SparkSession\
        .builder\
        .appName("Ethereum")\
        .getOrCreate()

    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url =
os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
    s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
    s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
    s3_bucket = os.environ['BUCKET_NAME']

    hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
    hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
    hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
    hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
    hadoopConf.set("fs.s3a.path.style.access", "true")
    hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")

    lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket +
"/ethereum-parvulus/transactions.csv")
    header = lines.first()
    clean_sample = lines.filter(lambda x: x != header)
    transactions = clean_sample.map(lambda x: (x.split(",")[6],
int(x.split(",")[7]))).reduceByKey(operator.add)

    # popular service
    join_lines = spark.sparkContext.textFile("s3a://" +
s3_data_repository_bucket + "/ethereum-parvulus/contracts.csv")
    c_header = join_lines.first()
    clean_contracts = join_lines.filter(lambda x: x != c_header)
    contracts = clean_contracts.map(lambda x: (x.split(",")[0], 1))
    joined_data = transactions.join(contracts)
    top10_popular_service = joined_data.takeOrdered(10, key=lambda x: -
x[1][0])
```

Firstly, the transactions dataset must be formatted such that the header is removed. The mapping function is then used to map the transaction `to_address` field from the csv file with its corresponding value (in Wei), the reduceByKey function then groups together the data and the final output shows the total value (Wei) for each Ethereum address. The contracts dataset is formatted and the mapping function maps the contract address field with the value 1. The join function will join the transaction RDD with the contracts RDD, then the takeOrdered function will extract the 10 highest transaction values by using the key parameter.

<u>Question C) Top Ten Most Active Miners</u>

The top ten most active miners by mining block size in bytes. The most a miner has mined is 17453393724.00 bytes or 17.45 Gigabytes (converted by dividing 10^9).





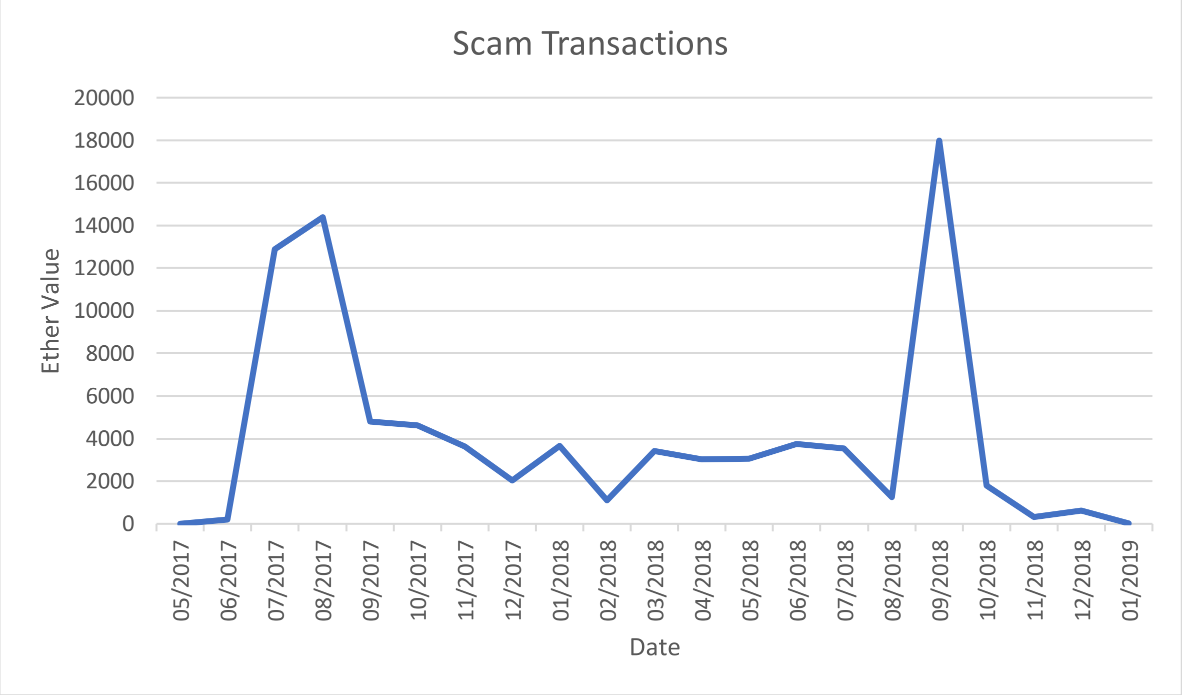
<u>Question C) Code for Top Ten Most Active Miners</u>

```python
import sys, string
import os
import socket
import time
import operator
import boto3
import json
import math
import time
import operator
from pyspark.sql import SparkSession
from datetime import datetime

if __name__ == "__main__":

    spark = SparkSession\
        .builder\
        .appName("Ethereum")\
        .getOrCreate()

    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url = 
os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
    s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
    s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
    s3_bucket = os.environ['BUCKET_NAME']

    hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
    hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
    hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
    hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
    hadoopConf.set("fs.s3a.path.style.access", "true")
    hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")

    block_lines = spark.sparkContext.textFile("s3a://" + 
s3_data_repository_bucket + "/ethereum-parvulus/blocks.csv")
    b_header = block_lines.first()
    clean_blocks = block_lines.filter(lambda x: x != b_header)
    top10_active = clean_blocks.map(lambda x: (x.split(",")[9], 
int(x.split(",")[12])))
    top10_active = top10_active.reduceByKey(operator.add).sortBy(lambda x: 
x[1], ascending=False)
```

Once the block dataset is formatted, the mapping function maps the miner address with the block size (in bytes) for each row, the reduceByKey function groups together the total block size for each miner. The sortBy function sorts the RDD by block size in descending order using the lambda function.

Question D) Data Exploration – Scam Analysis

The graph shows that overtime, the amount Ether lost to scams had varying peaks of high losses during 2017 and 2018



The scam **ID with the most lucrative scam is 2135** with 19110713823161336892288 Wei (19110.71 Ether)

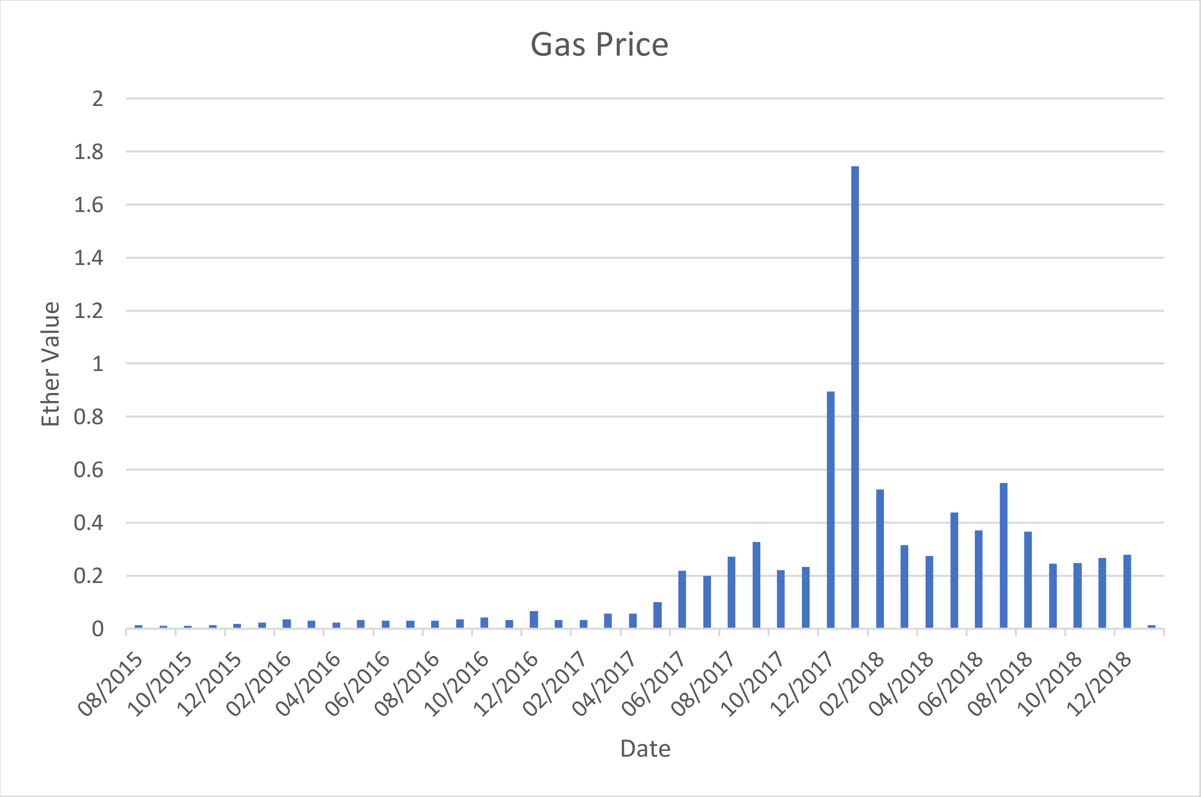
Question D) Code for Data Exploration – Scam Analysis

```python
import sys, string
import os
import socket
import time
import operator
import boto3
import json
import math
import time
import operator
from pyspark.sql import SparkSession
from datetime import datetime

if __name__ == "__main__":

    spark = SparkSession\
        .builder\
        .appName("Ethereum")\
        .getOrCreate()

    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url =
os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
    s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
    s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
    s3_bucket = os.environ['BUCKET_NAME']

    hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
    hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
    hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
    hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
    hadoopConf.set("fs.s3a.path.style.access", "true")
    hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")

    scam_data = spark.sparkContext.textFile("s3a://" +
s3_data_repository_bucket + "/ethereum-parvulus/scams.csv")
    scam_head = scam_data.first()
    scams = scam_data.filter(lambda x: x != scam_head)
    scam_map = scams.map(lambda x: (x.split(",")[6], int(x.split(",")[0])))

    lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket +
"/ethereum-parvulus/transactions.csv")
    header = lines.first()
    clean_sample = lines.filter(lambda x: x != header)
    transact = clean_sample.map(lambda x: (x.split(",")[6],
(datetime.fromtimestamp(int(x.split(",")[11])).strftime("%m/%Y"),
int(x.split(",")[7]))))
    join_up = transact.join(scam_map)

    scam_time = join_up.map(lambda x: (x[0],
x[1][0][1])).reduceByKey(operator.add)
    final_join = scam_time.join(scam_map)
    scamming = join_up.map(lambda x: (x[1][0][0],
x[1][0][1])).reduceByKey(operator.add)
    final_map = final_join.map(lambda x: (x[1][1],
x[1][0])).reduceByKey(operator.add).takeOrdered(10, key=lambda x: -x[1])
```

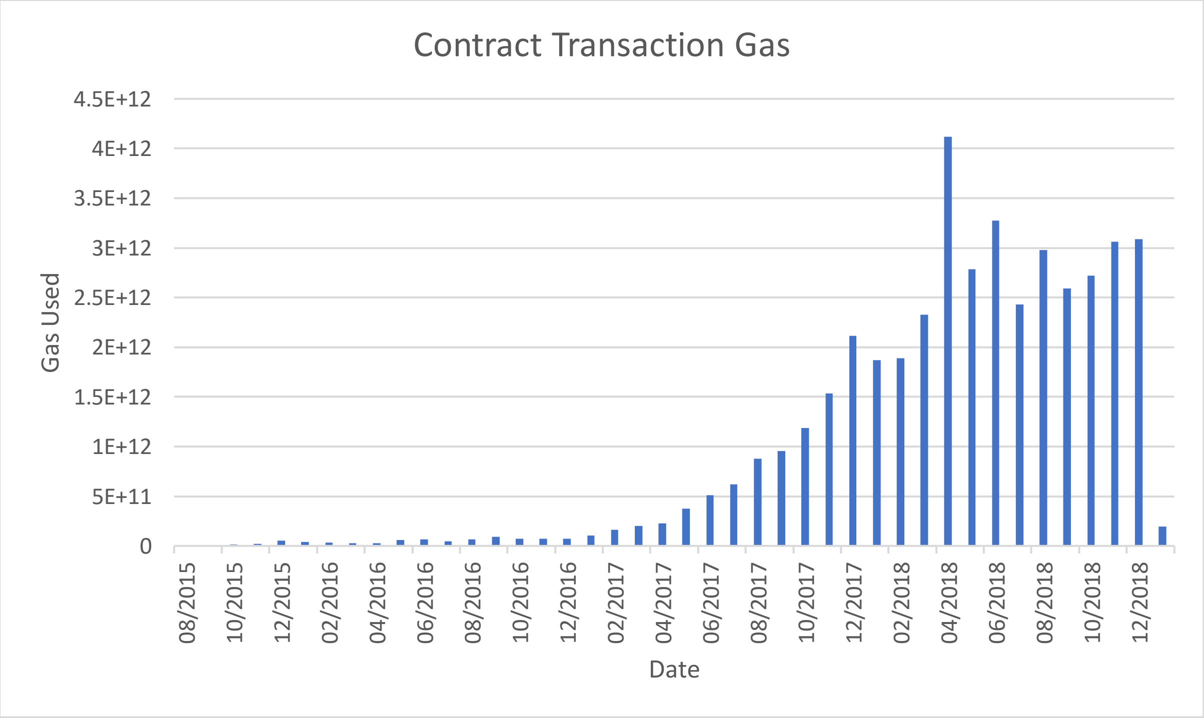
The scam dataset is formatted and the mapping function is used to map the Ethereum address with the scam ID. The transaction dataset is formatted and the mapping function is used to map the `to_address` field with the timestamp and transaction value (in Wei). A join function is used to join the two datasets and the mapping function will then map the Ethereum address with the transaction value. Another join function will then join the current mapping with the scam dataset. This would be used with another mapping and reduceByKey function to find the total value for each scam ID. Using the takeOrdered function will then give the most lucrative scam ID. Another mapping and reduceByKey function are used to get the total transaction value for each date, the scam dataset and transaction dataset are joined such that the output would be the total transaction value from the scams for each month.

Question D) Data Exploration – Gas Guzzlers

The **gas price (in Ether) over time** shows that in January 2018, the gas price was at its highest at 1.75 Ether.

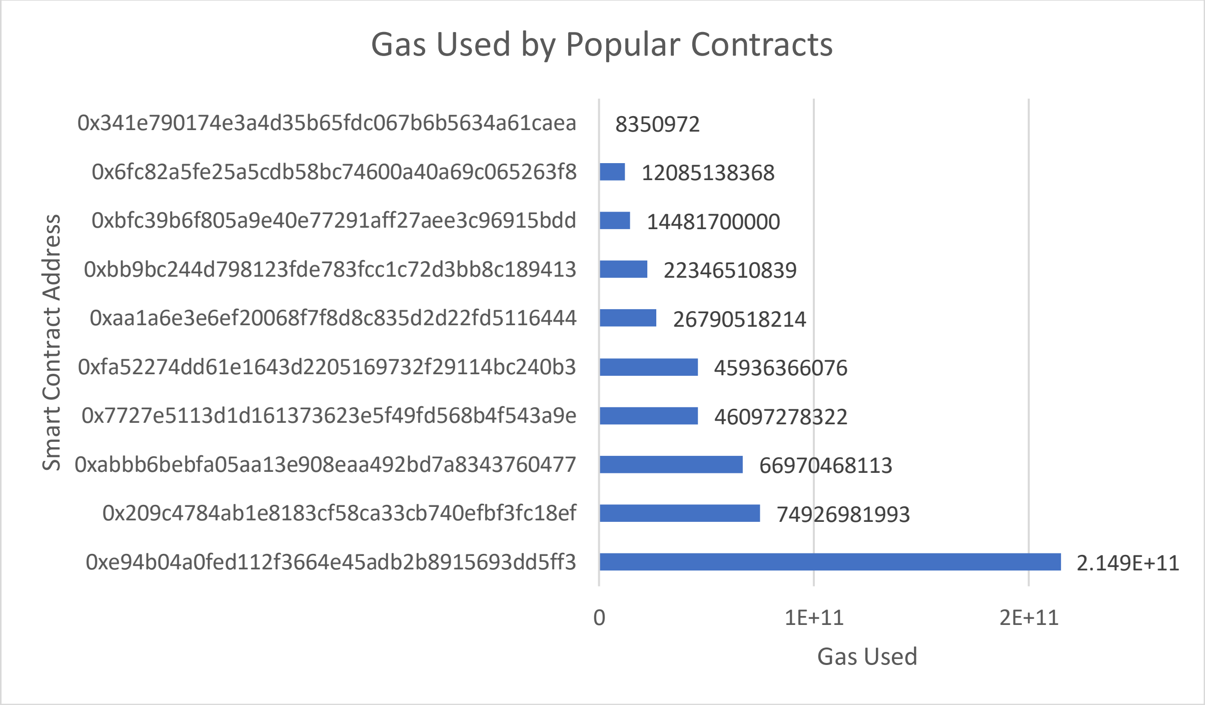


The **gas used for contract transactions over time** shows that in April 2018, 4.12E12 gas was used.



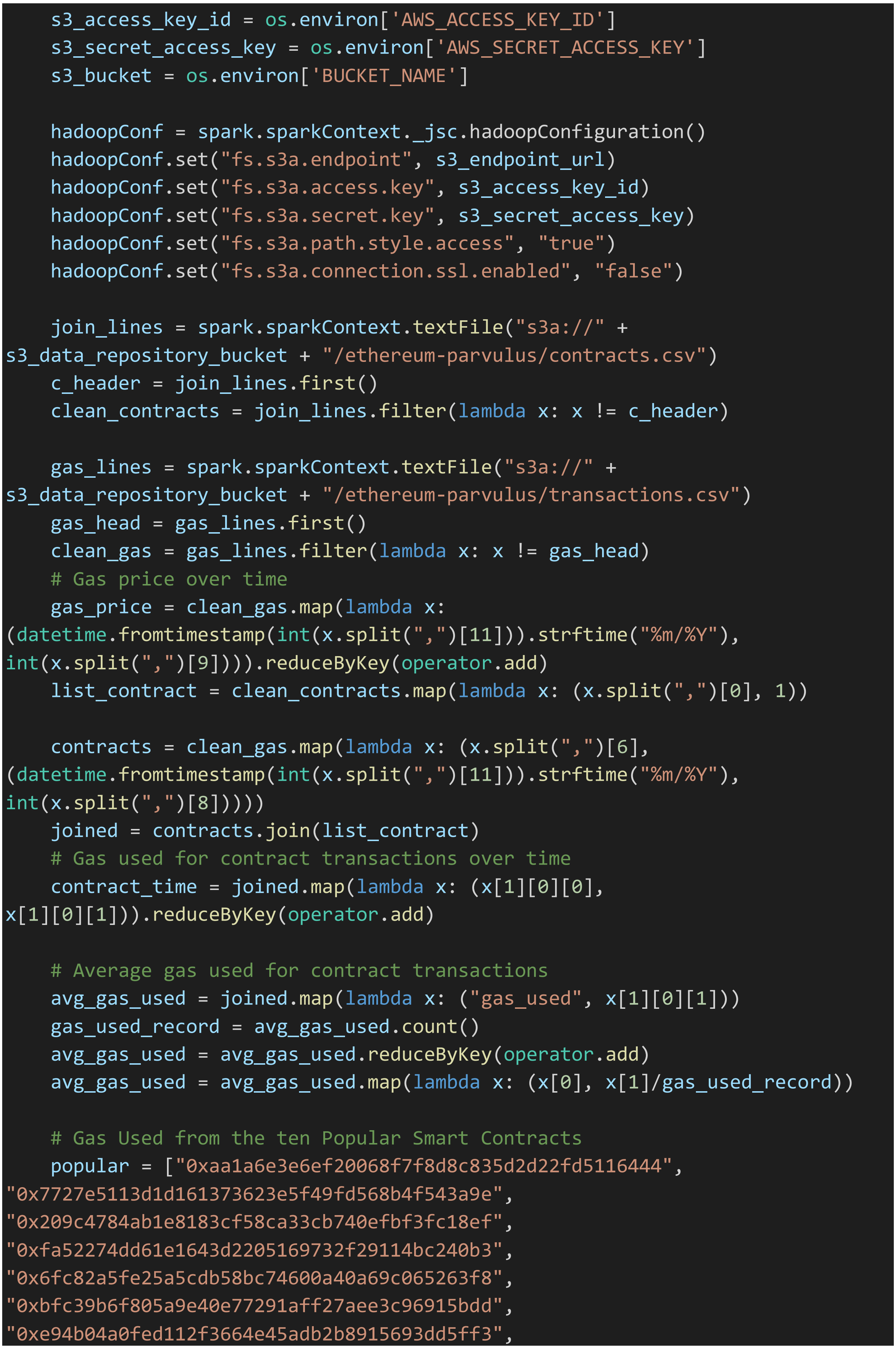
Average Gas Used: 263837.6006551491

The **gas used by the ten most popular smart contracts** shows that all the contracts use more than the average gas by a significant margin.



<u>Question D) Code for Data Exploration – Gas Guzzler</u>

```
import sys, string
import os
import socket
import time
import operator
import boto3
import json
import math
import time
import operator
from pyspark.sql import SparkSession
from datetime import datetime

if __name__ == "__main__":

    spark = SparkSession\
        .builder\
        .appName("Ethereum")\
        .getOrCreate()

    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url =
os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
```



```
    s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
    s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
    s3_bucket = os.environ['BUCKET_NAME']

    hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
    hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
    hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
    hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
    hadoopConf.set("fs.s3a.path.style.access", "true")
    hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")

    join_lines = spark.sparkContext.textFile("s3a://" +
s3_data_repository_bucket + "/ethereum-parvulus/contracts.csv")
    c_header = join_lines.first()
    clean_contracts = join_lines.filter(lambda x: x != c_header)

    gas_lines = spark.sparkContext.textFile("s3a://" +
s3_data_repository_bucket + "/ethereum-parvulus/transactions.csv")
    gas_head = gas_lines.first()
    clean_gas = gas_lines.filter(lambda x: x != gas_head)
    # Gas price over time
    gas_price = clean_gas.map(lambda x:
(datetime.fromtimestamp(int(x.split(",")[11])).strftime("%m/%Y"),
int(x.split(",")[9]))).reduceByKey(operator.add)
    list_contract = clean_contracts.map(lambda x: (x.split(",")[0], 1))

    contracts = clean_gas.map(lambda x: (x.split(",")[6],
(datetime.fromtimestamp(int(x.split(",")[11])).strftime("%m/%Y"),
int(x.split(",")[8]))))
    joined = contracts.join(list_contract)
    # Gas used for contract transactions over time
    contract_time = joined.map(lambda x: (x[1][0][0],
x[1][0][1])).reduceByKey(operator.add)

    # Average gas used for contract transactions
    avg_gas_used = joined.map(lambda x: ("gas_used", x[1][0][1]))
    gas_used_record = avg_gas_used.count()
    avg_gas_used = avg_gas_used.reduceByKey(operator.add)
    avg_gas_used = avg_gas_used.map(lambda x: (x[0], x[1]/gas_used_record))

    # Gas Used from the ten Popular Smart Contracts
    popular = ["0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444",
"0x7727e5113d1d161373623e5f49fd568b4f543a9e",
"0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef",
"0xfa52274dd61e1643d2205169732f29114bc240b3",
"0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8",
"0xbfc39b6f805a9e40e77291aff27aee3c96915bdd",
"0xe94b04a0fed112f3664e45adb2b8915693dd5ff3",
"0xbb9bc244d798123fde783fcc1c72d3bb8c189413",
"0xabbb6bebfa05aa13e908eaa492bd7a8343760477",
"0x341e790174e3a4d35b65fdc067b6b5634a61caea"]
    filt_contract = clean_contracts.filter(lambda x: x.split(",")[0] in
popular)
    new_list_contract = filt_contract.map(lambda x: (x.split(",")[0], 1))

    contracts = clean_gas.map(lambda x: (x.split(",")[6],
(datetime.fromtimestamp(int(x.split(",")[11])).strftime("%m/%Y")
,int(x.split(",")[8]))))
    new_joined = contracts.join(new_list_contract)

    popular_contract_gas = new_joined.map(lambda x: (x[0],
x[1][0][1])).reduceByKey(operator.add)
    print(popular_contract_gas.take(10))
```

For the transaction dataset, the mapping function is used to map the timestamp with the gas price and the reduceByKey function groups the data such that the RDD would show the total gas price (in Wei) for each month. The mapping function is then used to map the contract address from the contract dataset csv file to the value 1, this mapping would be used as part of a join operation. Another mapping function is used to map the transaction dataset address with its timestamp and the gas used, this mapping is then joined with the contract mapping to output a new RDD that shows only contract transactions. A map and reduce function are then used to get the total contract gas used for each month. The average gas used is calculated by using a map function and a constant string to map each gas used field to the same key. Once the count is recorded and the reduce function is executed, another mapping function is used to get the total average gas used. To calculate how much gas was used for each of the ten popular smart contracts, the address for each contract is stored in an array and the filter function is used to filter out the contract dataset for only the top ten popular contracts. Once the contract dataset is joined with the transaction dataset, a map and reduce function is used to get the total gas for each contract. This can then be compared with the total average gas used.

Question D) Data Exploration – Data Overhead

Total Byte Size for each column

The total bytes size of the redundant columns in the block dataset csv file is 21.5GB, the Logs_Bloom column takes up the most space at 1.79GB. Bits are converted to bytes by dividing by 8 and bytes are converted to GigaBytes by dividing 10^9

Columns in blocks.csv	Bits	Bytes	GigaBytes
Logs_Bloom	14336002048.00	1792000256.00	1.792000256
Sha3_Uncles	1792000256.00	224000032.00	0.224000032
Transactions_Root	1792000256.00	224000032.00	0.224000032
State_Root	1792000256.00	224000032.00	0.224000032
Receipts_Root	1792000256.00	224000032.00	0.224000032
Total	21504003072.00	2688000384.00	2.688000384

Total Storage Saved

The total file size of the block dataset is 154.9GB, once the redundant columns are removed, the new dataset size would be 152.2GB, thus 2.69GB is saved when the redundant columns are removed.

	Bytes	GigaBytes
Total Block File Size	154922109826.00	154.9221098
New Storage Size	152234109442.00	152.2341094
Amount Storage Saved	2688000384.00	2.688000384

Question D) Code for Data Exploration – Data Overhead

```
import sys, string
import os
import socket
import time
import operator
import boto3
import json
import math
import time
import operator
from pyspark.sql import SparkSession
from datetime import datetime

if __name__ == "__main__":

    spark = SparkSession\
        .builder\
        .appName("Ethereum")\
        .getOrCreate()

    s3_data_repository_bucket = os.environ['DATA_REPOSITORY_BUCKET']
    s3_endpoint_url =
os.environ['S3_ENDPOINT_URL']+':'+os.environ['BUCKET_PORT']
    s3_access_key_id = os.environ['AWS_ACCESS_KEY_ID']
    s3_secret_access_key = os.environ['AWS_SECRET_ACCESS_KEY']
    s3_bucket = os.environ['BUCKET_NAME']

    hadoopConf = spark.sparkContext._jsc.hadoopConfiguration()
    hadoopConf.set("fs.s3a.endpoint", s3_endpoint_url)
    hadoopConf.set("fs.s3a.access.key", s3_access_key_id)
    hadoopConf.set("fs.s3a.secret.key", s3_secret_access_key)
    hadoopConf.set("fs.s3a.path.style.access", "true")
    hadoopConf.set("fs.s3a.connection.ssl.enabled", "false")

    lines = spark.sparkContext.textFile("s3a://" + s3_data_repository_bucket +
"/ethereum-parvulus/blocks.csv")
```

```
    header = lines.first()
    clean_sample = lines.filter(lambda x: x != header)

    # Data Overhead
    # remove first 2 bits -> take size/length of hex string -> multiply by 4 -
> total number of bits
    # do instructions for logs_bloom, sha3_uncles, transactions_root,
state_root, receipts_root .map(lambda x: ("logs_bloom",
len(x.split(",")[5][2:])*4))
    # add all bytes up and get size of transaction file and calculate
difference
    logs_bloom = clean_sample.map(lambda x: ("logs_bloom",
len(x.split(",")[5][2:])*4)).reduceByKey(operator.add)
    sha3_uncles = clean_sample.map(lambda x: ("sha3_uncles",
len(x.split(",")[4][2:])*4)).reduceByKey(operator.add)
    transactions_root = clean_sample.map(lambda x: ("transactions_root",
len(x.split(",")[6][2:])*4)).reduceByKey(operator.add)
    state_root = clean_sample.map(lambda x: ("state_root",
len(x.split(",")[7][2:])*4)).reduceByKey(operator.add)
    receipt_root = clean_sample.map(lambda x: ("receipt_root",
len(x.split(",")[8][2:])*4)).reduceByKey(operator.add)

    file_size = boto3.session.Session().client(service_name="s3",
aws_access_key_id= os.environ['AWS_ACCESS_KEY_ID']+,
                                          aws_secret_access_key=
os.environ['AWS_SECRET_ACCESS_KEY']+,
                                          endpoint_url=
os.environ['S3_ENDPOINT_URL']).head_object(Bucket="data-repository-bkt",

                                      Key="/ethereum-
parvulus/transactions.csv")
    print(file_size['ContentLength'])
```

To calculate the size of the redundant columns in the block dataset csv file, the first two bits should be ignored, then the length of the hex string must be multiplied by 4 to get the total number of bits. For each redundant column, a mapping function is used where the key is a constant value, i.e., a string and the value is the corresponding bit size. The reduceByKey function would reduce and group together the total bit size for each column. To get the total file size of the block dataset, the AWS boto3 library is used where the credentials given can get the size of the file. Once the file size is retrieved, the calculations are made onto an excel spreadsheet and result is that 2.69GB is saved.