ECS640U Big Data Processing Coursework

December 6, 2022

0.0.1 Big Data Processing Coursework

Michael Peres

200362146 https://qmplus.qmul.ac.uk/mod/assign/view.php?id=2124104

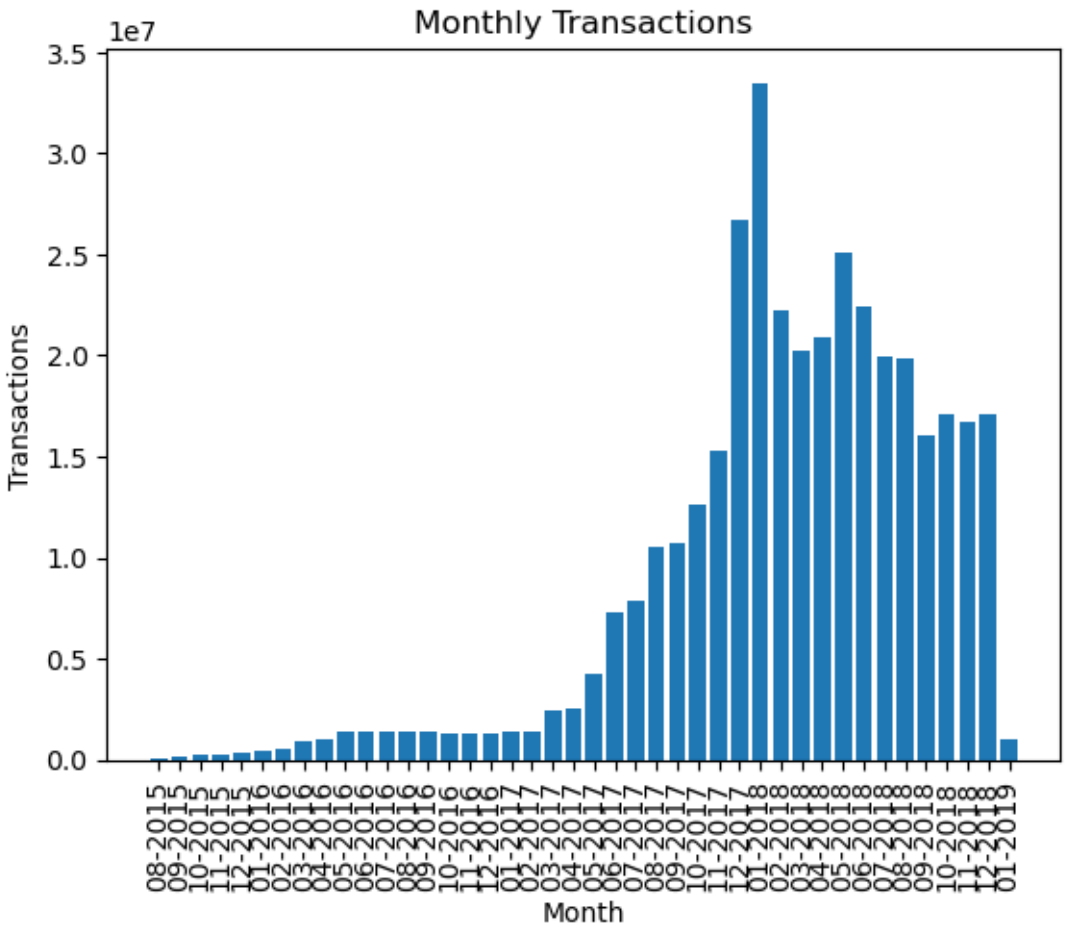
Part A. Time Analysis (25%) 1a) Create a bar plot showing the number of transactions occurring every month between the start and end of the dataset.

[1]:
```
import matplotlib.pyplot as plt
import json, os

# Read data from file
with open(os.path.join(os.getcwd(), "cw_output/monthly_transactions.txt"), "r")␣
 ↪as f:
    data = f.read().splitlines()

list_of_values = json.loads(data[0])  # ['08-2015', 85609] , , , , ...
x_values = [i[0] for i in list_of_values]  # dates listcomp
y_values = [i[1] for i in list_of_values]  # values listcomp

plt.bar(x_values, y_values, align="center")
# Title
plt.title("Monthly Transactions")
# X axis label
plt.xlabel("Month")
# Y axis label
plt.ylabel("Transactions")
# Show x labels clearly, by turning 90 degrees.
plt.xticks(x_values, rotation=90)
# Show plot
plt.show()
```



0.0.2 Explanation of code implementation in Part 1a

We obtain transactions from the s3 bucket.

Filter good lines by checking if line[11] can be converted to an int.

We then map this RDD by parsing its gm time, using transaction epoch obtained in line[11]

We then parse only the "%m-%Y" for example 01-2020 as a key for our reducer and the value 1.

In the reducer we simply add the values using operator.add

We then sort using sortBy function and base it on the key: datetime.strptime(date[0], '%m-%Y'), which is the key and the date string.

We then save this to a /monthly_transactions.txt file.

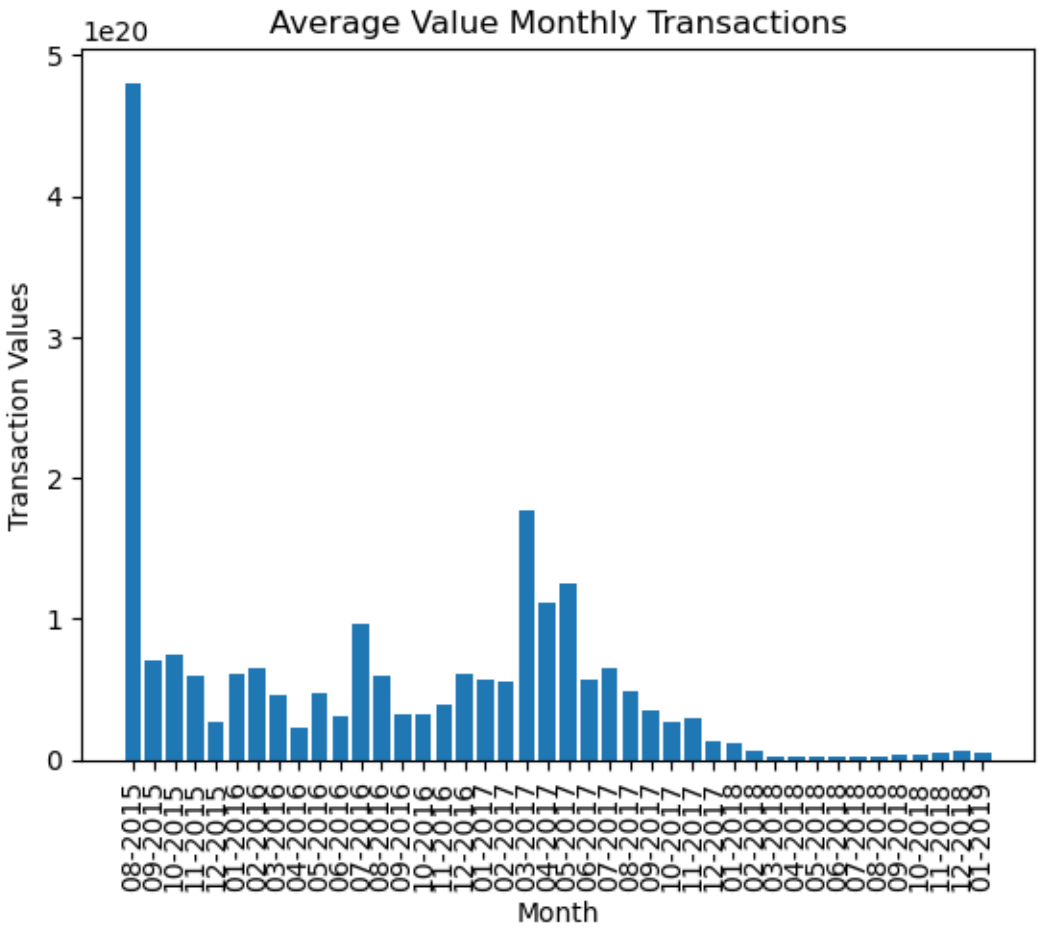
1b) Create a bar plot showing the average value of transaction in each month between the start and end of the dataset.

[9]:
```
import matplotlib.pyplot as plt
import json, os

# Read data from file
with open(os.path.join(os.getcwd(), "cw_output/avg_monthly_transactions.txt"),␣
 ↪"r") as f:
    data = f.read().splitlines()

list_of_values = json.loads(data[0])  # ['08-2015', 85609] , , , , ...
x_values = [i[0] for i in list_of_values]  # dates listcomp
y_values = [i[1] for i in list_of_values]  # values listcomp

plt.bar(x_values, y_values, align="center")
# Title
plt.title("Average Value Monthly Transactions")
# X axis label
plt.xlabel("Month")
# Y axis label
plt.ylabel("Transaction Values")
# Show x labels clearly, by turning 90 degrees.
plt.xticks(x_values, rotation=90)
# Show plot
plt.show()
```



0.0.3 Explanation of code implementation in Part 1b

We loaded the transaction data from s3 bucket, into a RDD.

We filter this data by looking if line[11] and line[7] are convertible to ints, this is the transaction epoch and the value of the transaction.

We map based on the first part, but now we also add to the value of the tuple the (1, value), count and value.

We then in the reducer, look at the date key, and add the counts and the value.

Given this we sortBy like part 1a, then map again, this time, dividing total value / total count, giving us average count.

We then return this to /avg_monthly_transactions.txt

Part B. Top Ten Most Popular Services (25%) Evaluate the top 10 smart contracts by total Ether received. You will need to join address field in the contracts dataset to the to_address

in the transactions dataset to determine how much ether a contract has received.

[DONE NEED TO WAIT FOR OUTPUT TO BE GENERATED AND SHOW PRESENTATION.]

[1]:
```
# Have done, however the code causes memory error when running.
# Will find answer and will be done with it.

import json, os
with open(os.path.join(os.getcwd(), "cw_output/top10.txt"), "r") as f:
    data = f.read().splitlines()

output = json.loads(data[0])
print("Top 10 addresses: ")
_ = [print(x[0]) for x in output]
```

```
Top 10 addresses:
0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444
0x7727e5113d1d161373623e5f49fd568b4f543a9e
0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef
0xfa52274dd61e1643d2205169732f29114bc240b3
0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8
0xbfc39b6f805a9e40e77291aff27aee3c96915bdd
0xe94b04a0fed112f3664e45adb2b8915693dd5ff3
0xbb9bc244d798123fde783fcc1c72d3bb8c189413
0xabbb6bebfa05aa13e908eaa492bd7a8343760477
0x341e790174e3a4d35b65fdc067b6b5634a61caea
```

0.0.4 Explanation of code implementation in Part B.

We read both contracts and transaction s3 files into RDDs.

We filter transactions, by whether value is convertible to a int, line[7] and if line[6] has a value or is None.

We then map transactions to obtain the format: (to_address, value) and reduce by using operator add.

We then map contracts to obtain the format: (address, bytecode)

We then join the transactions to contracts, using `transactions.join(contracts)`

We obtain the top10 using `top10 = top_set.takeOrdered(10, key=lambda x: -int(x[1][0]))`

Then we save it to /top10.txt

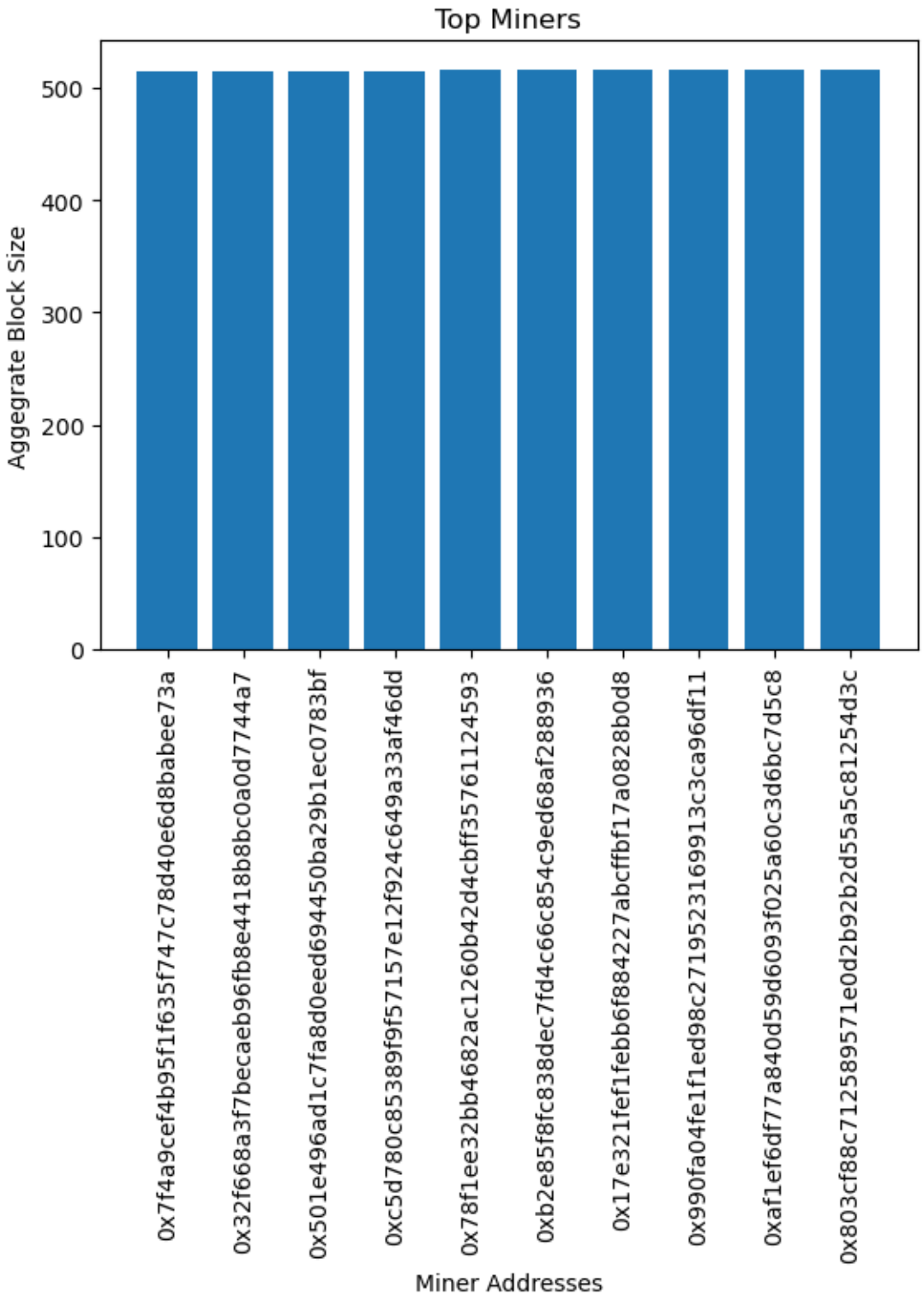
Part C. Top Ten Most Active Miners (10%) Evaluate the top 10 miners by the size of the blocks mined. This is simpler as it does not require a join. You will first have to aggregate blocks to see how much each miner has been involved in. You will want to aggregate size for addresses in the miner field. This will be similar to the word count that we saw in Lab 1 and Lab 2. You can add each value from the reducer to a list and then sort the list to obtain the most active miners.

[11]:
```
import matplotlib.pyplot as plt
import json, os

# Read data from file
with open(os.path.join(os.getcwd(), "cw_output/top_miners.txt"), "r") as f:
    data = f.read().splitlines()

list_of_values = json.loads(data[0])  # ['08-2015', 85609] , , , , ...
x_values = [i[0] for i in list_of_values]  # dates listcomp
y_values = [i[1] for i in list_of_values]  # values listcomp

plt.bar(x_values, y_values, align="center")
# Title
plt.title("Top Miners")
# X axis label
plt.xlabel("Miner Addresses")
# Y axis label
plt.ylabel("Aggegrate Block Size")
# Show x labels clearly, by turning 90 degrees.
plt.xticks(x_values, rotation=90)
# Show plot
plt.show()
```


Top Miners
Aggegrate Block Size
0
100
200
300
400
500
0x7f4a9cef4b95f1f635f747c78d40e6d8babee73a
0x32f668a3f7becaeb96fb8e4418b8bc0a0d7744a7
0x501e496ad1c7fa8d0eed694450ba29b1ec0783bf
0xc5d780c85389f9f57157e12f924c649a33af46dd
0x78f1ee32bb4682ac1260b42d4cbff35761124593
0xb2e85f8fc838dec7fd4c66c854c9ed68af288936
0x17e321fef1febb6f884227abcffbf17a0828b0d8
0x990fa04fe1f1ed98c2719523169913c3ca96df11
0xaf1ef6df77a840d59d6093f025a60c3d6bc7d5c8
0x803cf88c712589571e0d2b92b2d55a5c81254d3c
Miner Addresses

0.0.5 Explanation of code implementation in Part C.

We read block from s3 into a rdd.

We map the block by obtaining the miner and the size of block, line[9] and line[12].

We then reduce by operator.add, which gives us, (miner, total_size).

We obtain the top_miners by simply doing what we did in part b:

`top_miners = block.takeOrdered(10, key=lambda x: -x[1])` Here looking at key being the value negative to get top miners.

We then save this to /top_miners.txt

---

1 Part D. Data exploration (40%)

1.1 Scam Analysis: Wash Trading [25% Already Awarded]

We did attempt this code, and did finish it until the volume calculations,

This requires us to use the GraphFrame library which isn't available,

So Sir has awarded us 25% for this attempt of Wash Trading, everything up to Part 3 has been tested and works. Only sections including the use of GraphFrame cannot be tested, but theoretically works.

Explanation of Wash Trading Coursework: This is based on the paper, and so we go through 5 steps in order to determine likely suspects of wash trading. Our aim is to determine whether trading is occuring with no financial gain in the hopes the market seems to have much higher volume than its actuality.

The steps we go through to determine this are:

1.1.1 Part 1: Self trading filtering.

Here we attempt to see whether any individuals are trading with themselves, so this would be the case where `src` and `dst` address is the same.

We obtain there and put them in the /potential_self_wash_trading.txt file.

We do this by mapping, to obtain tuple `(from_address, [transaction_index, trans_value, timestamp])`. Which is returns to obtain: `(line1[0], [line1[1]+line2[1], line1[1].append(line2[1])])`

We filter if filtered_rdd has a key of none.

We save this as rdd filtered_self_transactions.

```
hashes_ids = filtered_self_transactions.map(lambda line: ("hashes",
[*line[1][1]])).reduceByKey(lambda x, y: x[0]+y[0]).collect()[0][1]
```

```
filtered_transactions = transactions_rdd.map(lambda line: line if line[0] not in
hashes_ids else ("none", None)).filter(lambda x: x[0] != "none")
```

We use these to remove these transactions from the initial transaction rdd. Self Join was a little complicated to do, so instead used map reduce filter paradigm and works equally as good.

1.1.2 Part 2: Possible Candidates.

Now that we have removed the self traders, based on lecturer advice we can now look at potential candidates for wash trading.

We first want to map, each address, with net value for the transaction.

We want to see if net volume for a specific address is 0, or within 3% of average transaction value.

We do this by flat mapping and return tuple `([from_addr, -int(value)], [to_addr, int(value)])`.

We then reduce by adding. Now that we have net transactions for each address, we know need to get average transaction value.

We do this mapping value and reducing by add, and then dividing by the count of filtered_transactions.

We then filter based on this, like this:

```
possible_candidates = net_transactions.filter(lambda x: abs(x[1]) < 0.15 *
avg_trans_value)
```

1.1.3 Part 3: Creating Graph Representation

Now we go into the fun part, which is trying to obtain a graph from our semi-processed data, for graph related calculations like RDD.

The graphframe library allows creations of Graphs but require us to use 2 specific RDDs, EDGES and VERTICES.

We create these using:

```
transaction_vertices = filtered_transactions.flatMap(vertices_map).reduceByKey(lambda
x, y: x)
```

```
transaction_edges = filtered_transactions.map(edge_map).reduceByKey(lambda x, y:
(x[1][0]+y[1][0], x[1][1]+y[1][1]))  # ((src,dst), (weight, value))
```

Given these RDDs, we create dataframes using the `createDataFrame` function listing our schema.

For the vertices, we will state schema to be: [“address”, “token_type”].

For the edges, we will state schema to be: [“src”, “dst”, “weight”, “value”].

We are able to then create graph like this:

```
graph = GraphFrame(transaction_vertices_dataframe, spark.createDataFrame(edges,
[("src", "dst"), ("weight", "value")]))
```

However we don't do this just yet.

1.1.4 Part 4: SCC Generation

A SCC is a strong connected component, we do this by creating a graph, and using an function:

```
sccs = graph.stronglyConnectedComponents(maxIter=10)
```

We simply, want to determine how many repeat SCCs there are, because if some group is trading multiple times, it can seem like it is suspicious and a likely candidate.

Our algorithm here, which is also explained in the paper referenced is to count SCC occurances, and reducing the edge weights at the same. This is better explained in the paper, but this is shown within my code, as an iteration,

The key points here within the iteration is that:

We create a graph every iteration with filtered edges and creating SCCs continously.

```
scc_vertexes = scc_rdd_vertex.map(lambda x: (x[3], [[x[0]],
0])).reduceByKey(lambda x, y: x[0][0] + y[0][0])  # [component, [address1,
address2, ...]]
```

```
scc_vertexes = scc_vertexes.map(scc_vertex_algo)  # [component, [address1,
address2, ...], weight]
```

We then filter edges based on the weight being more than 0. `edges = edges.rdd.map(edges_algo_map).filter(lambda x: x[1] > 0)`

Out of this we get a counting of (unqiue_scc, weight)

We set a arbitary threshold, say 3, that states if a SCC occurs more than 3 times it is considered suspicious.

```
THRESHOLD_SCC_COUNT = 3 candidate_set = scc_count.filter(lambda x: x[1] >=
THRESHOLD_SCC_COUNT)  # [[address1, address2, ...], weight]
```

We now have the possible candidate set, to confirm these sets are wash trading there is one last thing to do, check volume and position.

1.1.5 Part 5: Trade Volume Matching

Using the set of address we have now obtained, we will need to join to net_transactions we made earlier.

We want to see which cycle have a net 0 volume.

A sample psuedocode for this, as we haven't adequately tested and know the outputs of graph strongly connected dataframes is presented here:

```
sccs_layout = [[address1, address2, ...], weight, [edges]]
# volume_threshold = 0.01
# for scc in sccs:
    #  net_transactions = edges.join(transactions, scc_addresses, (src, address, "").map(net_t
    #  sum_transactions = net_transactions.map(lambda line: ("scc_sum", transaction_sum)).reduc
    # transaction_count = net_transactions.count()  # 3
    # avg_transaction = sum_transactions[0][1] / transaction_count  #  take scc_sum and divide
    # threshold = net_transactions.map(lambda line: ("scc_sum", abs(total_scc_value))).reduceBy
    # if abs(sum_transactions) <= threshold:
        # print("Wash trading detected")
        # collect in global rdd., if this exists
        # suspects_rdd = suspects_rdd.union(scc)
```

1.1.6 End of Wash Trading Explanation

1.2 Data Overhead 15% (v ez)

Result:

[10]:
```
import json, os
with open(os.path.join(os.getcwd(), "cw_output/data_overhead.txt"), "r") as f:
    data = f.read().splitlines()

output = json.loads(data[0])[0]
print(f"Output > Wasted Storage Used: {output[1]} bits")


def convert_bytes(size):
    for x in ['bytes', 'KB', 'MB', 'GB', 'TB']:
        if size < 1024.0:
            return "%3.1f %s" % (size, x)
        size /= 1024.0
    return size

print(f"Simplified Result: {convert_bytes(output[1]/8)}")
```

```
Output > Wasted Storage Used: 22422471840 bits
Simplified Result: 2.6 GB
```

Explanation of Data Overhead Coursework Here we read the blocks from s3 into a RDD.

We map this by initialising a local variable hex_count.

Obtain the variables within each line, for `sha3_uncles`, `logs_bloom`, `transaction_root`, `state_root`, `receipts_root` and `extra_data`.

We check if there is a string and starts with 0x, if so we obtain its length and remove 2 accounting for the “0x”

We know a character is equivalent to 4 bits, so we times by 4 and use in map tuple. Mapper key is “byte_count”

We simply add this at the reducer, and so all converge to a single reducer. We can use .collect() in memory, as we know only one tuple will return.

We save this to /data_overhead.txt.

The example result looks like this: [[“byte_count”, 22422471840]]

—— END ——