ECS640U/ECS765P - BIG DATA PROCESSING - 2021/22

COURSEWORK: ETHEREUM ANALYSIS

Name: Suleiman Suleiman
Student number: 160628964

Table of Contents

Ethereum Analysis

In this assignment I will be applying techniques learnt in the first half of my big data processing course to analyse the full set of transactions which have occurred on the Ethereum network; from the first transactions in August 2015 till June 2019. In all of the MapReduce jobs I used a combiner to speed up the sorting process.

Part A

In the first of part A, I created a bar chart to show the number of transactions that has occurred every month from the start to the end of the dataset. I used MapReduce to go through all these transactions so that it yields the number of transactions correlating to each date of the transaction in Month/Year format.

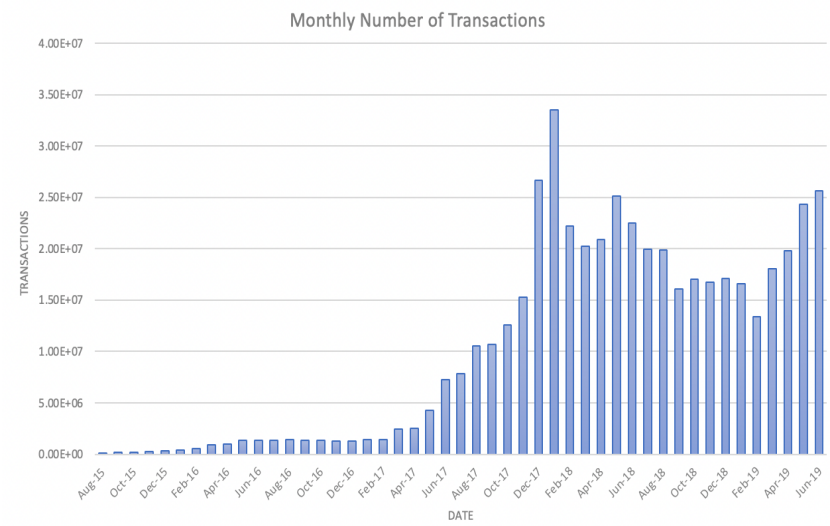


Figure 1: Monthly Number of Transactions graph

Code:

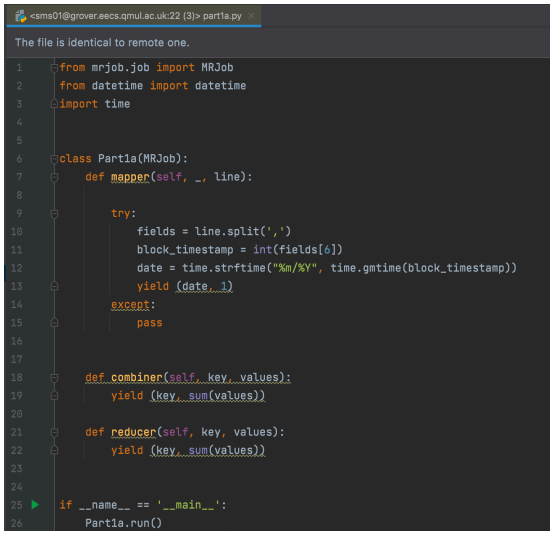


Figure 2: MapReduce for the number of transactions occurring every month

The mapper splits the lines then maps out each date a transaction has happened and gives it a value of 1. The date is given in Gregorian format using the time function. I used a method known as Try and Except in the mapper to ensure the code has no failures when I run it. The reducer then adds all the values emitted for every key that comes from the mapper. For every key the mapper produces the reducer would match the keys and add the values. For example if the reducer has 7 transactions that occurred in August 2015, each with a value of 1 given from the mapper, it will give an outcome of: |“August 2015” 7|

In the second part of part A, I created a bar chart to show the average value of transaction in each month between the start and end of the dataset. I used MapReduce to go through all these transactions so that it yields the average value in Wei correlating to that specific transaction.

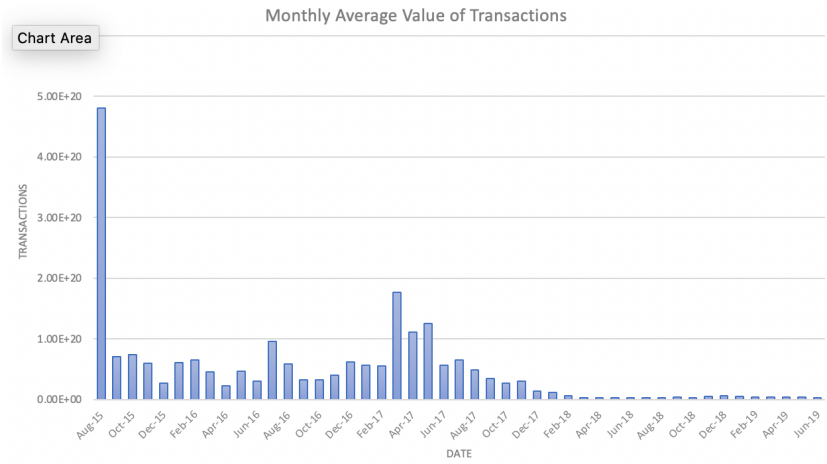


Figure 3: Monthly Average Value of Transactions graph

Code:

Figure 4:MapReduce for the value of transactions occurring every month

The mapper splits the line and maps out the date and the value of the transaction with a value of 1. The reducer then matches the keys and adds the values that was given by the mapper. The key, value pair that is fed into the reducer in this part is the date and the value of transaction. The reducer output is the date and the sum of the transaction values in that date range. A combiner was used to aid the reducer.

Part B

In this part of the assignment be I evaluated the top 10 smart contracts by total ether received.

Job1

In the first part of part B, I used MapReduce to workout which services are the most popular. This is done by aggregating **value** for addresses in the **to_address** field.

Below is a snapshot of a few lines of output from job1:

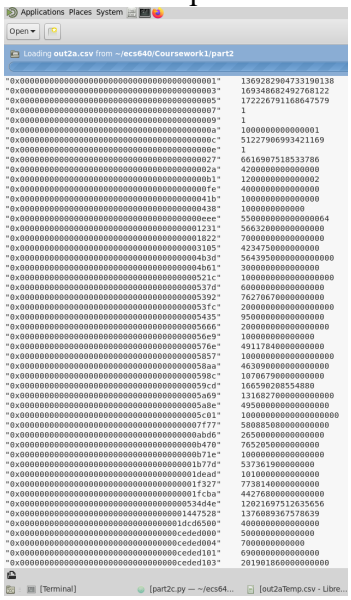


Figure 5: Snapshot from the output of job1

Code:

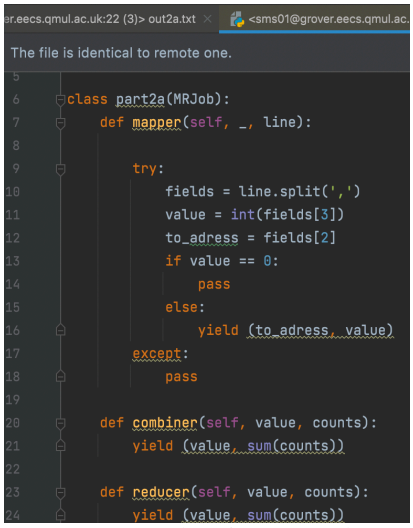


Figure 6: MapReduce for part B job1

The mapper splits the lines then it maps out the **to_address** and gives it a value depending on the value of that transaction. The reducer then uses the output from the mapper to match the **to_address** with the sum of values of the transaction.

Job2

In the second part of part B, I used MapReduce to join the **to_address** field from the output of Job 1 with the **address** field of **contracts.**

Below is a snapshot of a few lines of output from job2:

	A	B
1	0x0000000000000000000000000000000000000002	1.87872718893805E+016
2	0x0000000000000000000000000000000000000011	5200000000000
3	0x00000000000000000000000000000000000000fe	1000000000000000
4	0x0000000000000000000000000000000000005435	9.5E+016
5	0x00000000000000000000000000000000000060d4	1000000000000000
6	0x0000000000000000000000000000000000eeee	8.9E+018
7	0x0000000000000000000000012345432452323 4	1000000000000000
8	0x000000000000000000403f3bdfa635afc53ef8	1E+016
9	0x0000000000000000000c24ce3325b3a1f3f36c	1E+016
10	0x0000000000000000001f32dc2f5e747501ad22	1E+016
11	0x000000000000000000256d12a08e0d3b824c9c	3.5337887E+016
12	0x0000000000000000000287517dc618fe4090d7b	1E+016
13	0x000000000000000000003563ec21f7acfee3e45c	1E+016
14	0x0000000000000000003c2bf93d4a52a93d92b1	1E+016
15	0x0000000000000000003c9292c08c9622220105	1E+016
16	0x000000000000000000434e2f3bc90e627bc3b9	1E+016
17	0x0000000000000000004571fbc954dea9b3208f	1E+016
18	0x000000000000000000049a41aef1cc8fcdf4c23	1E+016
19	0x0000000000000000000719de0ad90d8e5a4a515	1E+016
20	0x000000000000000000008be0b7b59b8973dcc017	1E+016
21	0x0000000000000000008f7837d6d5a6db42f15f	1E+016
22	0x000000000000000000008fa8c14fe1b40c09012c	1E+016
23	0x000000000000000000000921e3e6a000fd61a1b1b	1E+016
24	0x000000000000000000000a2b9b312e4a0563d2225	1E+016
25	0x000000000000000000000a70bfbdd14a2b31f2df4	1E+016
26	0x000000000000000000000aaebe6daf326cbdace88	1E+016
27	0x000000000000000000000ac18771b9279d26a3690	1E+016
28	0x000000000000000000000af6cb7214d9ae1733606	1E+016
29	0x000000000000000000000c305ca649d4fb458d48a	1E+016
30	0x000000000000000000000d0e14a64fb2292b927c3	1E+016
31	0x000000000000000000000d93ec846014b97a5cfe8	1E+016
32	0x000000000000000000000daadcefff1c38fe11cdc	1E+016
33	0x000000000000000000000dc63c10e2b5c1f87ec4a	1E+016
34	0x000000000000000000000e93922b31f73f9d9af07	1E+016
35	0x000000000000000000000e9acc57644fc9bdb71c9	1E+016
36	0x000000000000000000000eba9d5b90d20822b2a97	1E+016
37	0x000000000000000000000bc6a57040a3a284f7017d	1000000000000000
38	0x000000004fdd41501a42651743c55f7e41ec85e4	2550518000000000
39	0x000000004b1ada7e713a9ffa5528c9fdda221fc11	8E+016
40	0x00000017f6f17ce18faa53d7010fa66acd0e1fb9	1E+016
41	0x0000001815d6f8920ea18fcdd72f7531b2d4e55a	3.9657788E+017
42	0x0000003c3cadc0357195a09138ace2f3c451bf61	3.5E+017
43	0x00000052ed7101e3182752b6be50733acfb5ba7d	5E+017
44	0x000006389361 6bfed0940aa7608e465e9fad8a4	8949827268333721
45	0x0000089398328f585811 9a3fb65f65c864c6dd	0
46	0x000000a541fa13beeb75f82a1663a3b532f8ada3e	9.43808341E+018
47	0x000000b3b21cfb4ae5646655dec51f6277c8ed5e	5E+017
48	0x000000c61e6a9f3b6a9d83d0c1922ed1a95a0f80	1.5E+017
49	0x000000c81745e30ac06838d7b81eca717f59e279	5E+017

Figure 7: Snapshot of output from job2

Code:

```
class part2b(MRJob):

    def mapper(self, _, line):
        try:
            if len(line.split(','))==5:
                fields = line.split(',')
                address = fields[0]
                yield (address,(address,1))

            if len(line.split(',')) == 7:
                fields = line.split(',')
                to_address = fields[2]
                value = int(fields[3])
                yield (to_address, (value,2))
        except:
            pass
    def reducer(self, key, vals):
        block_number = None
        total = 0
        for val in vals:
            if val[1] == 1:
                block_number = val[0]
            if val[1] == 2:
                total = val[0]

        if block_number is None and total >= 0:
            yield (key, total)
```

Figure 8: MapReduce for part B job2

In the mapper I differentiate the 2 input files by checking the number of fields in the dataset. If there are 5 fields the contracts dataset will be used and if there are 7 fields the transactions dataset will be used. The key produced by the mapper in the contracts dataset is the block address. They key and value pair produced by mapper from the transactions dataset is the same key and value pair produced by the mapper in job1. The output from this mapper will be the ones with both keys matching which filters out the smart contracts. The reducer used a for loop to identify the aggregate values from the mappers output. If val[1] == 2 then **total** is the aggregate value. The output of the reducer was the smart contract address (key) and the aggregate value (value).

Job3

In the final part of part B, I used MapReduce to find the top 10 aggregate values. This job used the results from job1 and sorted using a top 10 reducer.

Below is a snapshot of the output from job3:

```
1       "0xaa1a6e3e6ef20068f7f8d8c835d2d22fd511644 - 841551008099658658227267 76"
2       "0xfa52274dd61e1643d2205169732f29114bc240b - 45787484483189352986478805"
3       "0x7727e5113d1d161373623e5f49fd568b4f543a9 - 45620624001350712557268573"
4       "0x209c4784ab1e8183cf58ca33cb740efbf3fc18e - 43170356092262468919298969"
5       "0x6fc82a5fe25a5cdb58bc74600a40a69c065263f - 27068921582019542499882877"
6       "0xbfc39b6f805a9e40e77291aff27aee3c96915bd - 21104195138093660050000000"
7       "0xe94b04a0fed112f3664e45adb2b8915693dd5ff - 15562398956802112254719409"
8       "0xbb9bc244d798123fde783fcc1c72d3bb8c18941 - 11983608729202893846818681"
9       "0xabbb6bebfa05aa13e908eaa492bd7a834376047 - 11706457177940895521770404"
10      "0x341e790174e3a4d35b65fdc067b6b5634a61cae - 83790007519177556240575 00"
```

Figure 9: Output of the top 10 smart contracts

Code:

```
class Part2c(MRJob):

    def mapper(self, _, line):
        try:
            fields = line.split()
            if len(fields) ==2:

                to_address = fields[0]
                value = int(fields[1])
                row = (to_address, value)
                yield(None, row)

        except:
            pass


    def combiner(self , key, lines):
        sorted_values = sorted(lines, reverse=True, key=lambda tup: tup[1])
        top = sorted_values[:10]

        for top_10 in top:
            yield (None, top_10)


    def reducer(self, key, lines):
        sorted_values = sorted(lines, reverse=True, key=lambda tup: tup[1])
        top = sorted_values[:10]

        for top_10 in top:
            yield (None, top_10)
```

Figure 10: MapReduce code for top 10 smart contracts

I used the results obtained in job2 to as the dataset for job 3. The mapper splits the lines and yields the sum of aggregate values. The key is none because the value needs to be sorted based on aggregate value. In the combiner the values are sorted in ascending order by using **reverse=True** which then passes through a for loop to iterate through the sorted values to yield the sorted values in ascending order. The reducer then yields the top 10 smart contracts using the for loop to iterate through the values using the same operation as the combiner. The for loop only iterates through the first 10 sorted values.

Part C

In this part of the assignment, I used MapReduce to find the most active miners using the size of the block mined. The data is taken from the block dataset.

Below is a snapshot of the output for the job in part C:

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

Figure 11: Output of the top 10 miners

Code:

```
def mapper(self, _, line):
    try:
        fields = line.split(',')
        if len(fields) == 9 :
            miner = fields[2]
            size = int(fields[4])
            yield (miner, size)

        else:
            pass

    except:
        pass


def reducer_1(self, key , size):
    yield (None,(key,sum(size)))


def reducer_2(self,key ,value):
    sorted_values = sorted(v, reverse=True, key=lambda tup: tup[1])
    i = 0
    for value in sorted_values:
        yield (key, value)
        i += 1
        if i >= 10:
            break



def steps(self):
    return [MRStep(mapper=self.mapper,
                   reducer_1=self.reducer_1),
            MRStep(reducer=self.reducer_2)]
```

Figure 12: MapReduce to find top 10 miners

The mapper splits the lines and yields the sum of aggregate values. The key is none because the value needs to be sorted based on aggregate value. In the combiner the values are sorted in ascending order by using **reverse=True** which then passes through a for loop to iterate through the sorted values to yield the sorted values in ascending order. The reducer then yields the top 10 smart contracts using the for loop to iterate through the values using the same operation as the combiner. The for loop only iterates through the first 10 sorted values

Part D

Gas Guzzlers

In this part of the report I will use MapReduce to aggregate the monthly gas fees in the transaction dataset to see how gas fees have varied over time.

Below is a snapshot of the output of how gas price has varied overtime:

```
"2015-08"   13675526628145667
"2015-11"   12583476104430362
"2016-01"   22911076013331960
"2016-03"   30080460339750914
"2016-05"   31981390916556827
"2016-07"   30706184556173272
"2016-09"   35060460913157551
"2016-10"   42705203278890015
"2016-12"   66225169253198349
"2017-02"   32497701028484277
"2017-04"   56781256271069560
"2017-06"   218784602251092080
"2017-08"   272611078122291462
"2017-11"   234162338645463319
"2018-01"   1745775524213227428
"2018-03"   315067212049139490
"2018-05"   437404482697267097
"2018-07"   548389573884093767
"2018-09"   244285201300061576
"2018-10"   247784878897508715
"2018-12"   279518438392310163
"2019-02"   388206277862784215
"2019-04"   229430905789955949
"2019-06"   386154104347355990
"2015-09"   9821946760863180
"2015-10"   11052272460855430
"2015-12"   19402278511760992
"2016-02"   35976721417127390
"2016-04"   23900730327612119
"2016-06"   31114050018381134
"2016-08"   31484196041993591
"2016-11"   32063652665727275
"2017-01"   31728112295093517
"2017-03"   56372389627027613
"2017-05"   100076640370730336
```

Figure 13: Snapshot of output of gas price over time

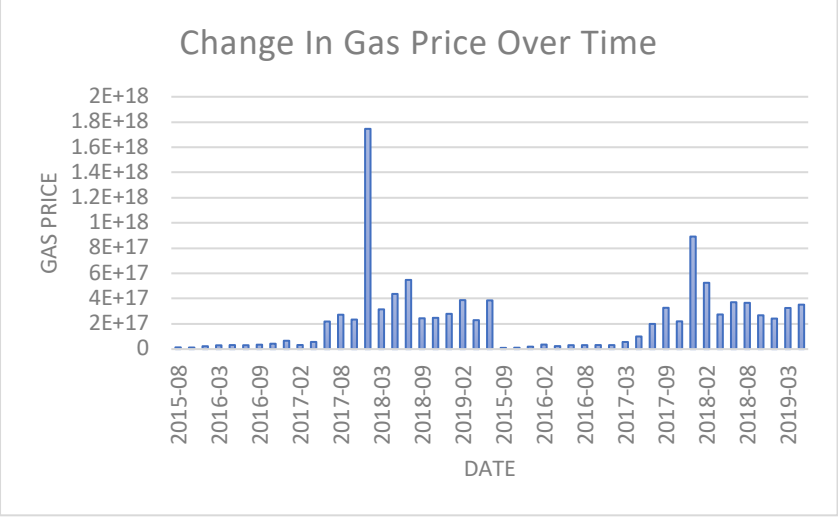


Figure 14: Graph showing Change In Gas Price Over Time

From the results obtained I can see that gas fees have been rising over time. A deduction can be made, comparing the graph in *figure1,* that as the number of transaction has increased so has the gas price.

Code:

```python
from mrjob.job import MRJob
from datetime import datetime
class gas(MRJob):
    def mapper(self, _, line):
        try:
            fields = line.split(',')
            if len(fields) == 7:
                block_timestamp = int(fields[6])
                date = time.strftime("%m/%Y", time.gmtime(block_timestamp))
                gas_price = int(fields[5])
                yield (date, gas_price)
        except:
            pass

    def combiner(self, key, value):
        yield (key, sum(value))

    def reducer(self, key, value):
        yield (key, sum(value))


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

Figure 15: MapReduce for monthly gas aggregate

In the mapper the key is the **date** and the value for each key is the **gas_price**. The combiner is used to sort the output from the mapper and calculate the sum of gasses for each key. The reducer uses the same logic as the combiner and yields the total **gas_price** for every month of the dataset.

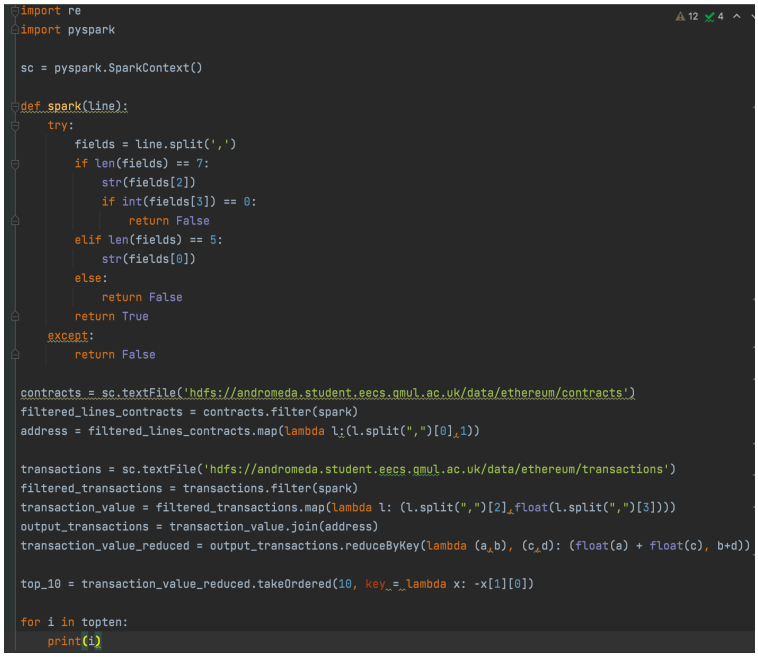
Spark

In this part of the assignment I done part B again using sparks. I used a single spark job to do all the jobs done in part B. the first step was to aggregate the **value** of the transactions in the transactions dataset. The next part was to filter the smart contracts out and to exclude the user address. The last step was to sort the addresses In ascending order then to filter out the top 10 smart contracts.

```
(u'0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444', (8.415510080995875e+25, 322149))
(u'0xfa52274dd61e1643d2205169732f29114bc240b3', (4.578748448318687e+25, 1491246))
(u'0x7727e5113d1d161373623e5f49fd568b4f543a9e', (4.562062400134807e+25, 513002))
(u'0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef', (4.317035609226541e+25, 1811324))
(u'0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8', (2.7068921582017373e+25, 289823))
(u'0xbfc39b6f805a9e40e77291aff27aee3c96915bdd', (2.1104195138094485e+25, 362046))
(u'0xe94b04a0fed112f3664e45adb2b8915693dd5ff3', (1.556239895680292e+25, 1616593))
(u'0xbb9bc244d798123fde783fcc1c72d3bb8c189413', (1.198360872920253e+25, 173561))
(u'0xabbb6bebfa05aa13e908eaa492bd7a8343760477', (1.1706457177941025e+25, 710462))
(u'0x341e790174e3a4d35b65fdc067b6b5634a61caea', (8.379000751917755e+24, 42))
```

Figure 16: snapshot of the output from part B using spark

Code:



```
import re
import pyspark

sc = pyspark.SparkContext()

def spark(line):
    try:
        fields = line.split(',')
        if len(fields) == 7:
            str(fields[2])
            if int(fields[3]) == 0:
                return False
        elif len(fields) == 5:
            str(fields[0])
        else:
            return False
        return True
    except:
        return False

contracts = sc.textFile('hdfs://andromeda.student.eecs.qmul.ac.uk/data/ethereum/contracts')
filtered_lines_contracts = contracts.filter(spark)
address = filtered_lines_contracts.map(lambda l:(l.split(",")[0],1))

transactions = sc.textFile('hdfs://andromeda.student.eecs.qmul.ac.uk/data/ethereum/transactions')
filtered_transactions = transactions.filter(spark)
transaction_value = filtered_transactions.map(lambda l: (l.split(",")[2],float(l.split(",")[3])))
output_transactions = transaction_value.join(address)
transaction_value_reduced = output_transactions.reduceByKey(lambda (a,b), (c,d): (float(a) + float(c), b+d))

top_10 = transaction_value_reduced.takeOrdered(10, key = lambda x: -x[1][0])

for i in topten:
    print(i)
```

Figure 17: Snapshot of the spark job used for part B

From completing the job on spark and Hadoop I have come to conclusion that the spark job was quicker than all 3 of the jobs using Hadoop and anytime the job had to repeat, the spark job was quicker. The spark job was not only less time consuming in running it, but it was also less time consuming when writing it. When using spark the entire execution was done in 1 job however when using Hadoop at least 2 jobs were needed.