ECS765P - Big Data Processing

Student Name: Nirmalkumar Pajany **Student ID: 210239797**

ANALYSIS OF ETHEREUM TRANSACTIONS AND SMART CONTRACTS

PRELUDE

All the tasks given for the coursework shall be done in both Hadoop and Spark. Hadoop is a batch processing framework. It solves the required problem by following the Map-Reduce principles. Data is converted into key-value pairs in Map stage by mappers. The yielded key-value pairs are shuffled, sorted and then given to reducers to reduce them by further processing the data if needed. The use of combiners after the map stage, which does almost the same job as the Reducer, is capable of fastening the map reduce framework. All these things are done parallely in multiple nodes.

Spark could easily overshadow the capabilities of Hadoop's map reduce in many dimensions. One important dimension remains to be the pace at which Spark does the job. Apart from this, Spark supports RDD (Resilient Distributed Data), DataFrame and DataSet. RDD is the oldest, while DataSet is the latest. All these DataStructures are immutable. While Hadoop supports only mappers, combiners and reducers spark supports the same and also many others under the two operations: Transformations and Actions. One great advantage is that the transformations are executed only when needed, i.e, only on the invocation of the actions the execution chain starts its work.

For the purpose of learning I chose Spark. The tasks are solved using RDDs and DataFrames. We are using pyspark python package for Spark

Pyspark methods used in this course work:

- filter(func f): Iterates over every row in RDD/DataFrame and filters them based on the condition/function passed
- map(func f): Iterates over every row in RDD/DataFrame to apply the given function
- reduceByKey(func f): Merges the values for each key, performs the given function on the values.
- join(to_join): This function is called on the data which needs to be joined and the other data which we need to join is passed as argument. The data shall be in RDD or in DataFrame
- takeOrdered(num): Returns N elements from the RDD in ascending order
- withColumn(): Used on DataFrames to manipulate DataFrame columns. To be noted that this method returns a new DataFrame instance since DataFrames are immutable
- sort(): Used on DataFrames to sort the rows based on a particular column

SparkContext creation:

```
sc = pyspark.SparkContext()
```

SparkSession Creation:

```
spark = SparkSession.builder.appName("Apps By NK").getOrCreate()
```

SOLUTIONS

PART A: Time Analysis

App ID: application_1649894236110_5588

Transactions table has been used for this task. Blocks table could also be used to calculate average transactions of a month, but we need to calculate average values for each month too. So I have used the transactions table.

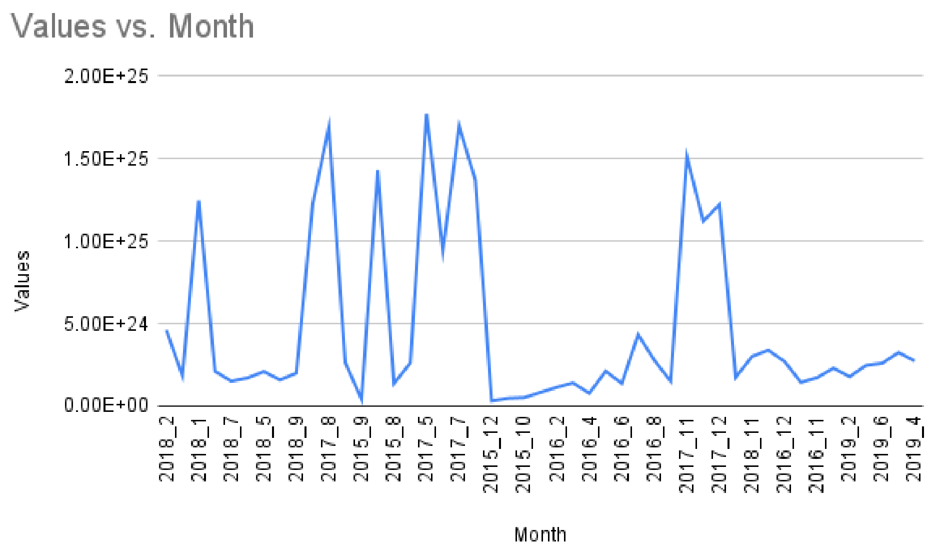


Fig: A.1: Average value each month

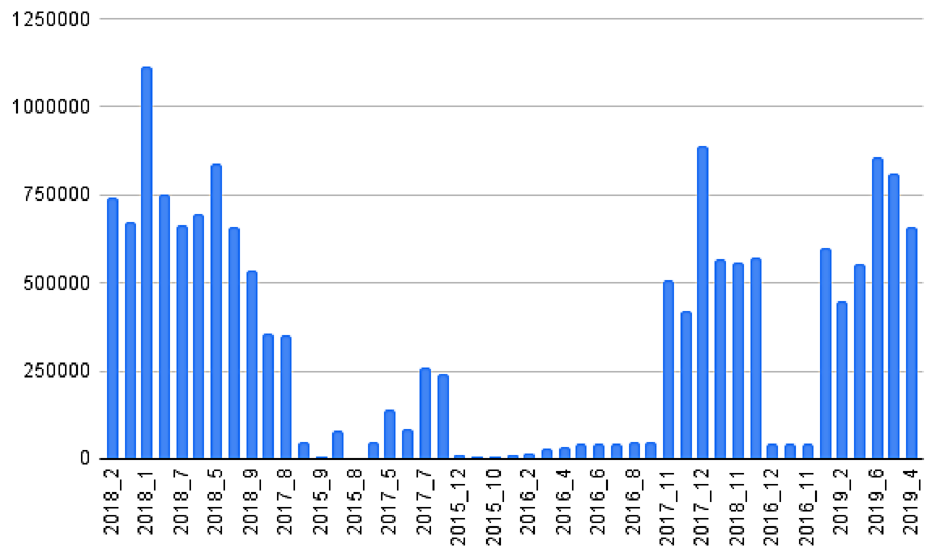


Fig: A.2: Average transactions each months

Code Snippet:

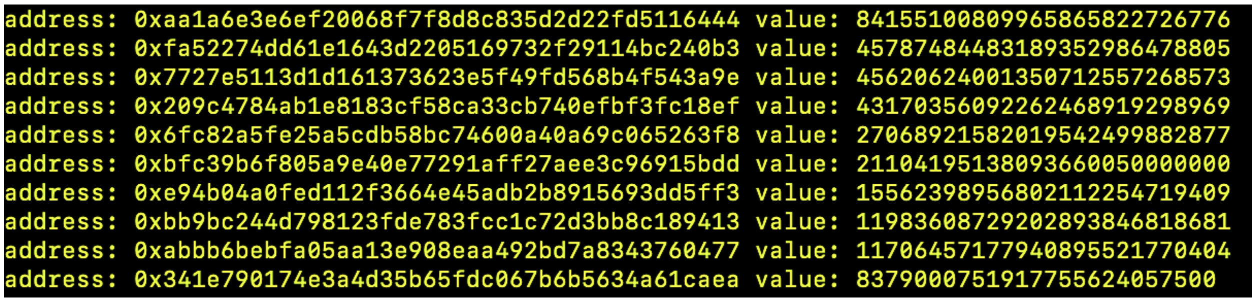
```
def get_features(line):
    try:
        fields = line.split(',')
        dd_mm = date.fromtimestamp(int(fields[6]))
        dd_mm = str(dd_mm.year) + '_' + str(dd_mm.month)
        val = int(fields[3])
        r_t = (dd_mm, (val, 1))
        return r_t
    except:
        return (0, (1, 1))

lines = sc.textFile("/data/ethereum/transactions")

clean_lines = lines.filter(is_clean)

features = clean_lines.map(get_features)

grouped_keys = features.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))
```

The above code first filters the data. The function is_clean is used to reject malformed csv lines. Next for each row in the file, we extract the month from its timestamp and return it as a tuple as described in the code. We then merge all the keys and while doing so we sum up app the respective values. The average is calculated separtely while plotting the graph.

PART B: Top Ten Most Popular Services

App ID: application_1649894236110_5581

As said in the Task description, first the values column of the transactions table has been aggregated based based on the to_address. Then this has then been joined with the contracts table to extract the top ten miners. The tope ten miners are as follows:



```
address: 0xaa1a6e3e6ef20068f7f8d8c835d2d22fd5116444 value: 84155100809965865822726776
address: 0xfa52274dd61e1643d2205169732f29114bc240b3 value: 45787484483189352986478805
address: 0x7727e5113d1d161373623e5f49fd568b4f543a9e value: 45620624001350712557268573
address: 0x209c4784ab1e8183cf58ca33cb740efbf3fc18ef value: 43170356092262468919298969
address: 0x6fc82a5fe25a5cdb58bc74600a40a69c065263f8 value: 27068921582019542499882877
address: 0xbfc39b6f805a9e40e77291aff27aee3c96915bdd value: 21104195138093660050000000
address: 0xe94b04a0fed112f3664e45adb2b8915693dd5ff3 value: 15562398956802112254719409
address: 0xbb9bc244d798123fde783fcc1c72d3bb8c189413 value: 11983608729202893846818681
address: 0xabbb6bebfa05aa13e908eaa492bd7a8343760477 value: 11706457177940895521770404
address: 0x341e790174e3a4d35b65fdc067b6b5634a61caea value: 8379000751917755624057500
```

Code Snippet:

```python
def get_features(line):
    try:
        fields = line.split(',')
        if fields[3] != 'value':
            to = str(fields[2])
            values = int(fields[3])
            return (to, values)
        else:
            return ('tmp', 0)
    except:
        return None

def get_contract_features(line):
    try:
        fields = line.split(',')
        if len(fields) == 5 and fields[3] != 'block_number':
            to = str(fields[0])
            return (str(to), 1)
        else:
            return (str('tmp_v'), 1)
    except:
        return (str('tmp_v'), 1)

transactions = sc.textFile("/data/ethereum/transactions")

contracts = sc.textFile("/data/ethereum/contracts")
```

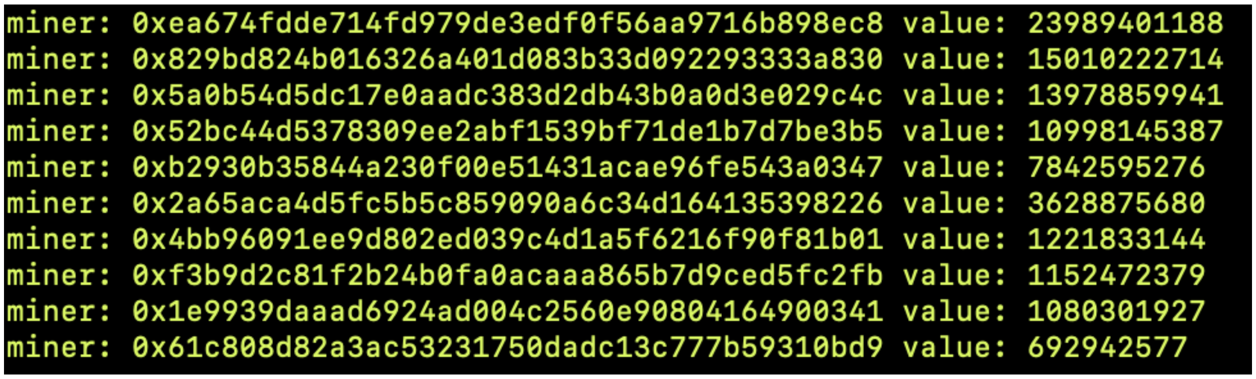
```
cleaned_cons = contracts.filter(clean_cons)

trans_as_keys = trans_features.reduceByKey(lambda a, b: a + b)

con_reduced = con_features.reduceByKey(lambda a, b: a + b)

joined = trans_as_keys.join(con_reduced)

top10 = joined.takeOrdered(10, key=lambda l: -l[1][0])
```

The above code first filters and then extracts the features from both the tables. Then they are reduced with reduceByKey function and when reducing they are aggreggated as per the need. Now the aggregated values are joined using the pyspark join function. We are performing inner join by using the to_address from transactions table and address from the contracts table. The code for all the three jobs are in the same file

PART C: Top Ten Most Active Miners

App ID: application_1649894236110_5778

This is a straightforward task handling only the Blocks table. Top Ten miners as follows:



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

Code Snippet:

```
def get_features(line):
    try:
        fields = line.split(',')
        miner = str(fields[2])
        size = int(fields[4])
        return (miner, size)
    except:
        return (0, 1)


blocks = sc.textFile("/data/ethereum/blocks")

cleaned_blocks = blocks.filter(is_clean)

block_features = cleaned_blocks.map(get_features)

miner_values = block_features.reduceByKey(lambda a, b: a + b)

top10 = miner_values.takeOrdered(10, key=lambda l: -l[1])
```

We extract the features and then straightly reduce them based on their keys. Then we extract the top ten miners based on their size using the takeOrdered function. Note that since this function sorts in ascending order we are ascending on the negative values.

PART D: Data Exploration

I Popular Scams:

App ID: application_1649894236110_5851

For popular scams I have matched the addresses from the scams.json file with the addresses in the Transactions table. Here by address I mean both the from and to address because when a scam happens we don't who does it. I have then aggregated their values to come up with top 5 as follows (Values are normalised by 10,00,000 for the purpose of inference):

Category	**Status**	**Value**
Scamming	Offline	36083529100
Scamming	Offline	60000000000
Scamming	Offline	990340000
Phising	Offline	109500100
Phising	Offline	9989500000

From the above table it could be clearly inferred that the most lucarative category is **Scamming** and Status is **offline**.

Code Snippet:

```
def load_json():
    print("App ID: {}".format(spark.sparkContext.applicationId))
    return spark.read.option("multiline", "true").json(json_path)

def load_transaction():
    trans_schema = StructType([StructField("block_number", StringType(), True),
                               StructField("from_address", StringType(), True),
                               StructField("to_address", StringType(), True),
                               StructField("value", LongType(), True),
                               StructField("gas", StringType(), True),
                               StructField("gas_price", StringType(), True),
                               StructField("block_timestamp", StringType(),
True)])
    return spark.read.option("multiline",
"true").schema(trans_schema).csv(csv_path)


def prep_json(scamsDF):
    try:
        count = scamsDF.count()
        rows = scamsDF.head(count)
        scams_list = []

        for r in rows:
            scam_dict = r['result'].asDict()
            keys = scam_dict.keys()
            for k in keys:
                scam = scam_dict[k].asDict()
                cat = str(scam['category'])
                stat = str(scam['status'])
                adds = scam['addresses']
                for a in adds:
                    t = (str(a), cat, stat)
                    scams_list.append(t)

        scams_schema = StructType([StructField("address_id", StringType(),
True),
                                   StructField("category", StringType(), True),
                                   StructField("status", StringType(), True)])

        return spark.createDataFrame(data=scams_list, schema=scams_schema)
    except:
        return None
```

```
jsonFile = load_json()
csvDF = load_transaction()
jsonDF = prep_json(jsonFile)

joinedDF = jsonDF.join(csvDF, jsonDF.address_id == csvDF.from_address, 'inner')
joinedDF = joinedDF.withColumn("value", col("value").cast(LongType()))
table = joinedDF.sort("value", ascending=False).take(100)
```

I have used DataFrames for this problem. The scams.json file is loaded using the spark.read() method. The the json is preprocessed in the prep_json method. To create DataFrames I have defined StructType. The Dataframes are joined using the join method, the data type of "value" is column is casted to LongType and the same is used to sort. The sorted values are saved anmd then I extracted the top ten values seperately. The most important thing for this program is that all these were done with **SparkSession** while the rest others are done with SparkContext.

II Fork the Chain:
App ID: application_1648683650522_6544
App ID: application_1648683650522_6556
App ID: application_1648683650522_6620

For this task I have choosen the following forks: i) Byzantium Fork, ii) Spurious Dragaon and iii) Tangerine Whistle
The are as follows observations:

Name	**avg_trans**	**avg_value**	**avg_gas**	**Period**
Tangerine Whistle	42291	40768159405610743510	131185	before
Tangerine Whistle	42659	40131965517404759559	104022	after
Byzantium Fork	321825	32099969234724551818	145001	before
Byzantium Fork	476539	24848456348832002968	145156	after
Spurious Dragon	41909	39966825268623882507	104201	before
Spurious Dragon	44467	47704290514035164421	136040	after

The calculations is the table are averaged for a month before and after the fork. From the table: Tangerine Whistle have had the lowest impact in all the three sectors while the Spurious Drangon has had bigger impacts on all the three. Byzantium

Fork has disturbed only the number transactions and value but not average gas consumed.

Code Snippet:

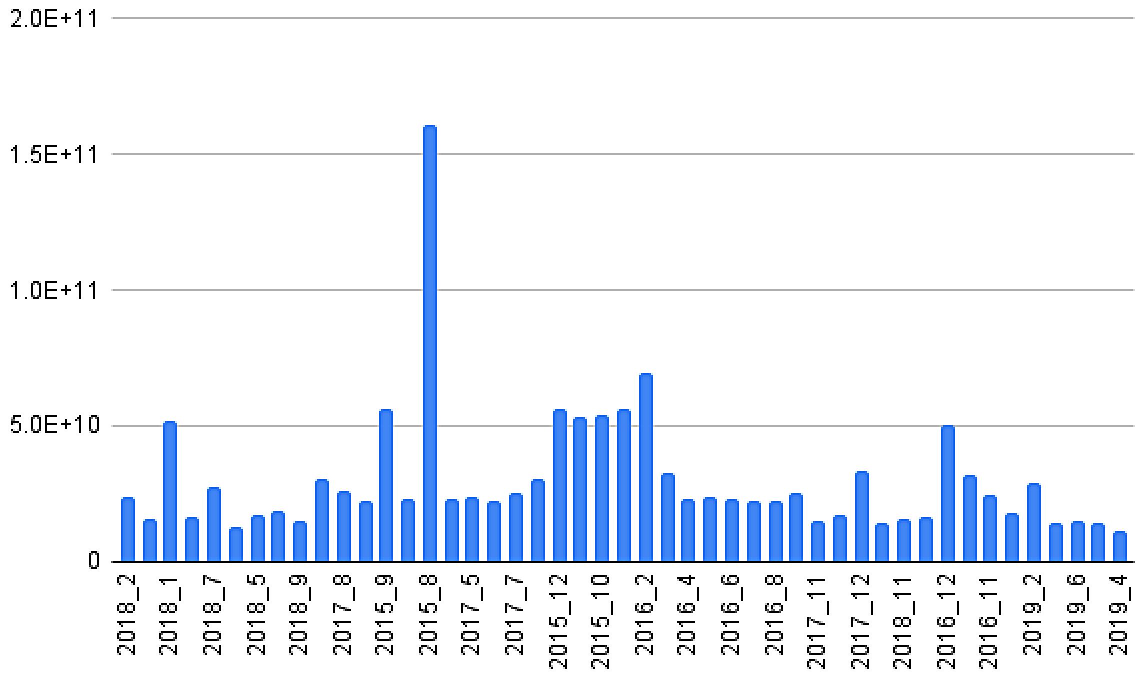
```
def get_features(line):
    try:
        fields = line.split(',')
        time = int(fields[6])
        value = int(fields[3])
        gas_price = int(fields[4])
        if time >= m_b and time <= forked_at:
            return (0, (1, value, gas_price))
        elif time <= m_a and time >= forked_at:
            return (1, (1, value, gas_price))
    except:
        return (9999, (1, 1, 1))


def cal_avg(line):
    try:
        k = int(line[0])
        total_trans = int(line[1][0])
        total_value = int(line[1][1])
        total_gas = int(line[1][2])

        avg_trans = total_trans / 31
        avg_value = total_value / total_trans
        avg_gas = total_gas / total_trans
        return (k, (avg_trans, avg_value, avg_gas))
    except:
        return (9999, (1, 1, 1))

transactions = sc.textFile("/data/ethereum/transactions")

clean_trans = transactions.filter(clean_transactions)

trans_features = clean_trans.map(get_features)

trans_as_keys = trans_features.reduceByKey(lambda a, b: (a[0] + b[0], a[1] +
b[1], a[2] + b[2]))

final_values = trans_as_keys.map(cal_avg)
```

The program is simple as the other programs mentioned above. The same program was executed three different times, for three different timestamps. This is not a necessary one, but I enjoyed running it thrice!!!

iii) Gas Guzzlers:

App ID: application_1648683650522_6656

For this task, transactions table alone shall be used. I happened to first convert the timestamps of each entry in transactions table into a month format and utilised the gas value for that entry. Later these values where merged based on the months and then averaged for the number of transactions happening on that month.



From the above gas we coudl infer that the August month of 2015 has seemed the highest average cost for the gas.

Code Snippets:

```
def get_features(line):
    try:
        fields = line.split(',')
        dd = date.fromtimestamp(int(fields[6]))
        dd_mm = str(dd.year) + '_' + str(dd.month)
        gas = int(fields[5])
        return (dd_mm, (gas, 1))

    except:
        return (0, (gas, 1))

def cal_avg(line):
    try:
        key = line[0]
        total_gas = int(line[1][0])
        total_counts = int(line[1][1])

        return (key, float(total_gas / total_counts))
    except:
        return (0, 0)

transaction = sc.textFile("/data/ethereum/transactions")

clean_trans = transaction.filter(clean_transaction)

features = clean_trans.map(get_features)

grouped_keys = features.reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1]))

final_values = grouped_keys.map(cal_avg)
```

The code is straight forward implenting the logic as described above.

iv) **Comparitive Evaluation:**

Since I was solving all the tasks in spark I used MRJob for this task. MRJob is a python package for hadoop support. One thing that I inferred from this task is that the time taken for execution in spark far exceeds hadoop's. Programmatically spark has more advantages. It is easy to implement the logic and the transactions and actions are not restricted to mapper and reducer alone. In spark the execution pipeline starts only when it is needed there by saving a lot of resources. The major data structures or records that we use in Spark (RDD, DataFrame and DataSet) are immutable which make perfect sense since we are parallelising our work in multiple nodes. In spark the execution flow is defined in a lineraly fashion and we don't need to flex more the choose the order. But for hadoop, especially for MRJob we need to overload MRStep to define the order of execution. The program for the MRJob version is attached in the src folder of my submission folder.

CONCLUSION

This was a great hands on learning experience with live clusters. I was able to clearly understand how computational needs shall be parallelized and a lot of time shall be saved. The work done so far was just to analyse huge data sets, while these same shall be used to train huge machine learning models too.