Scripts Execution

Explanation of the solution to the streaming layer problem

- All the necessary libraries and functions have been imported:

```
#all the necessary libraries and functions have been imported:

from pyspark.sql import SparkSession
from pyspark.sql functions import *
from pyspark.sql functions import *
from pyspark.sql.types import *
from datetime import datetimme
```

- Spark session is created and all the required python files are imported:

```
spark = SparkSession.builder.appName("Capstone_Project").getOrCreate()
spark.sparkContext.setLogLevel('ERROR')
sc = spark.sparkContext

# Adding required files
sc.addPyFile('db/dao.py')
sc.addPyFile('db/geo_map.py')
sc.addFile('rules/rules.py')

import dao
import geo_map
import rules
```

To connect to Kafka use the following details:
- Bootstrap-server: 18.211.252.152
- Port Number: 9092
- Topic: transactions-topic-verified

- The session is now connected to the server “18.211.252.152:9092” and to kafka topic "transactions-topic-verified

```
lines = spark.readStream.format("kafka") \
    .option("kafka.bootstrap.servers", "18.211.252.152:9092") \
    .option("subscribe", "transactions-topic-verified") \
    .option("failOnDataLoss", "false") \
    .option("startingOffsets", "earliest") \
    .load()
```

- The schema is defined and parsed in the format:

```
#Defining Schema

schema = StructType([
            StructField("card_id", StringType()),
            StructField("member_id", StringType()),
            StructField("amount", IntegerType()),
            StructField("pos_id", StringType()),
            StructField("postcode", StringType()),
            StructField("transaction_dt", StringType())
])

#parsing the data
parse = lines.select(from_json(col("value") \
                               .casr("strinf")  \
                                ,schema).alias("parsed"))

#parsing the dataframe
df_parsed = parse.select("parsed.*")
```

- The “look_up_table” and “card_transactions” table for card transaction details are used
- A set of user defined functions are used in order to perform the required activates and to check whether the transactions are fraudulent or genuine

- Function for fetching the Credit Scores from the look up tables using the “card_id”

```
#Adding Time Stamp Column
df_parsed = df_parsed.withColumn('transaction_dt_ts', unix_timestamp(df_parsed.transaction_dt, 'dd-MM-YYYY HH:mm:ss').cast(TimestampType()))

#Funciton for Credit Score
def score(a):
    hdao = dao.HbaseDao.get_instance()
    data_fetch = hdao.get_data(key=a, table='look_up_table')
    return data_fetch['info:score']

#defining UDF for Credit Score
score_udf = udf(score, StringType())

#adding score column
df_parsed = df_parsed.withColumn("score", score_udf(df_parsed.card_id))
```

- A function for fetching postal code from the look up tables using the “card_id”

```
#function for Postal Code

def postcode(a):
    hadao = dao.HBaseDao.get_instance()
    data_fetch = hdao.get_data(key=a, table='look_up_table')
    return data_fetch['info:postcode']

#defining UDF for Postal Code
postcode_udf = udf(postcode, StringType())

#Adding Postal Code Column
df_parsed = df_parsed.withColumn("last_postcode", postcode_udf(df_parsed.card_id))
```

- Function to fetch Upper Control Limit from the look up tables using the “card_id”

```
#Function for UCL

def ucl(a):
    hdao = dao.HBaseDao.get_instance()
    data_fetch = hdao.get_data(key=a, table='look_up_table')
    return data_fetch['info:UCL']

#defining UDF for UCL
ucl_udf = udf(ucl, StringType())

#adding UCL column
df_parsed = df_parsed.withColumn("UCL", ucl_udf(df_parsed.card_id))
```

- Function to calculate the distance between previous and current transaction postal codes from the look up tables and kafka stream

```
# Function for calculating distance
def dist_cal(last_postcode, postcode):
    gmap = geo_map.GEO_Map.get_instance()
    last_lat = gmap.get_lat(last_postcode)
    last_longg = gmap.get_long(last_postcode)
    lat = gmap.get_lat(postcode)
    longg = gmap.get_long(postcode)
    d = gmap.distance(
        last_lat.values[0],
        last_longg.values[0],
        lat.values[0],
        longg.values[0]
    )
    return d

# Defining UDF for Distance
distance_udf = udf(dist_cal, DoubleType())

# Adding distance column
df_parsed = df_parsed.withColumn("distance", distance_udf(df_parsed.last_postcode, df_parsed.postcode))
```

- Function to fetch the last transactions date from the look up tables using the “card_id”

```
#function for transaction date

def Tdate(a):
    hdao = dao.HBaseDao.get_instance()
    data_fetch = hdao.get_data(key=a, table='look_up_table')
    return data_fetch['info:transaction_date']

#defining UDF for transaction date
Tdate_udf = udf(Tdate, StringType())

#adding transaction date column
df_parsed = df_parsed.withColumn("last_transaction_date", Tdate_udf(df_parsed.card_id))
```

- Function to calculate the time difference (in sec) between the previous and current transactions from the look up tables and kafka stream

```
#Adding the timestamp column

df_parsed = df_parsed.withColumn('last_transaction_date_ts',
                                 unix_timestamp(df_parsed.last_transaction_date,
                                                'dd-MM-YYYY HH:mm:ss').cast(TimestampType()))


def time_cal(last_date, curr_date):
    d = curr_date - last_date
    return d.total_seconds()


#defining UDF for calculating time
time_udf = udf(time_cal, DoubleType())

#adding time difference column
df_parsed = df_parsed.withColumn("time_diff", time_udf(df_parsed.last_transaction_date_ts, df_parsed.transaction_dt_ts))
```

- The function to define the status of transaction is fraudulent or genuine

```
#Function to define the status of the transaction

def status_def(card_id, member_id, amount, pos_id, postcode, transaction_dt, transaction_dt_ts,
               last_transaction_date_ts, score, distance, time_diff):
    hdao = dao.HBaseDao.get_instance()
    geo  = geo_map.GEO_Map.get_instance()
    look_up = hdao.get_data(key=id_card, table='look_up_table')
    status = 'FRAUD'
    if rules.rules_check(data_fetch['info:UCL'], score, distance, time_diff, amount):
        status = 'GENUINE'
        data_fetch['info:transaction_date'] = str(transaction_dt_ts)
        data_fetch['info:postcode'] = str(postcode)
        hdao.write_data(card_id, data_fetch, 'look_up_table')
    row= {'info:postcode': bytes(postcode), 'info:pos_id': bytes(pos_id), 'info:card_id': bytes(card_id), 'info:amount': bytes(amount),
          'info:transaction_dt': bytes(transaction_dt), 'info:member_id': bytes(member_id), 'info:status': bytes(status)}
    key = '{0}.{1}.{2}.{3}'.format(card_id, member_id, str(transaction_dt), str(datetime.now())).replace(" ", "").replace(":", "")
    hdao.write_data(bytes(key), row, 'card_transactions')
    return status


#defining UDF for status
status_udf = udf(status_find, StringType())

#Adding status column
df_parsed = df_parsed.withColumn('status', status_udf(df_parsed.card_id, df_parsed.member_id,
                                                      df_parsed.amount, df_parsed.pos_id, df_parsed.postcode, df_parsed.transaction_dt,
                                                      df_parsed.transaction_dt_ts, df_parsed.last_transaction_date_ts, df_parsed.score,
                                                      df_parsed.distance, df_parsed.timme_diff))
```

- Function to define the rules to check if the transaction is genuine or fraudulent.
    1. Transaction amount < UCL
    2. Time difference in sec < 4 times the distance
    3. Credit Score < 200

```
#list all the funcitons to check for the rules

#fucntion to define rules

def rules_check(UCL, score, distance, time_diff, amount):
    if amount < UCL:
        if time_diff < (distance*4):
            if score > 200:
                return True
    return False
```

- Displaying the required columns in the console and the query has to be terminated properly

```
#printing Output on console

query1 = df_parsed  \
        .writeStream    \
        .outputMode("append")   \
        .format("console")  \
        .option("truncate", "False")    \
        .start()

#terminating the query
query1.awaitTermination()
```

- Use PuTTY to connect to the initialized EC2 instance and log in as the root user.
- Access the HBase Shell to locate the "card transactions" and lookup tables required for the driver.py script, then exit back to the root user.
- Create a directory named CapStone and place the following files within it:
    - *driver.py*
    - *geomap.py*
    - *dao.py* (in a subdirectory named db)
    - *rules.py* (in a subdirectory named rules)
    - *uszipsv.csv*

- In **dao.py**, update self.host to 'localhost' for execution purposes.

- Ensure the Thrift server is up and running.

- Confirm the happybase module is installed by running:

```
python -c 'import happybase'
```

- Run the following commands as the root user:
  - Download the required Spark-Kafka JAR file

```
wget https://ds-spark-sql-kafka-jar.s3.amazonaws.com/spark-sql-kafka-0-10_2.11-2.3.0.jar
```

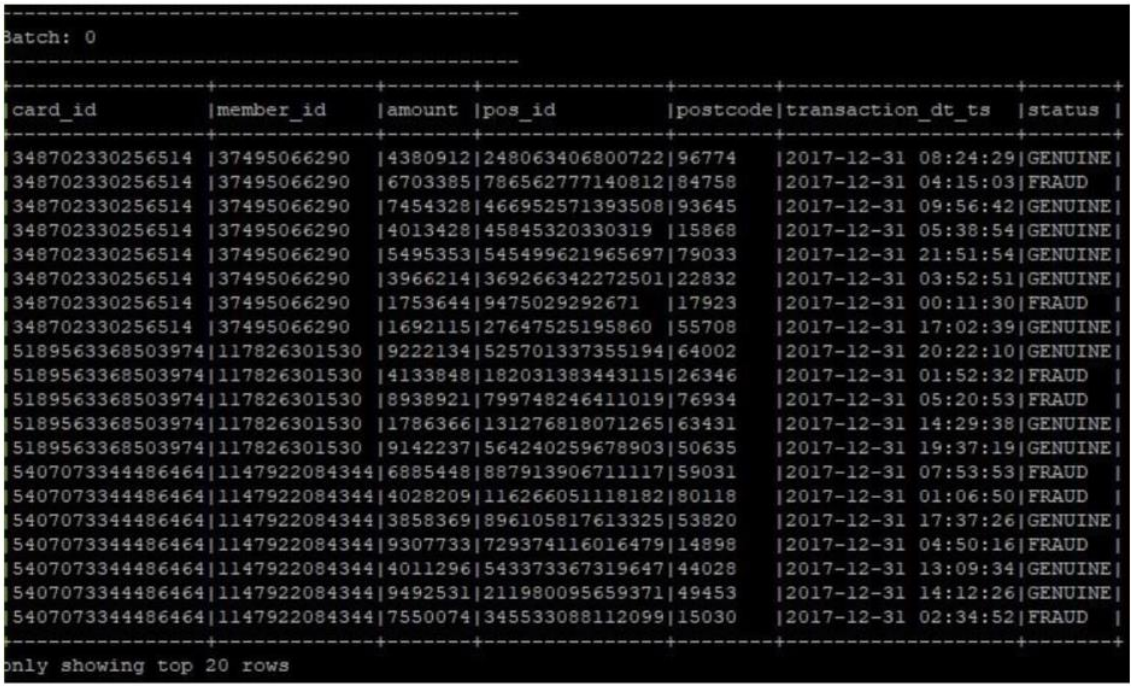
  - Set the Kafka version environment variable:

```
export SPARK_KAFKA_VERSION=0.10
```

  - Execute the **driver.py** script with Spark, using the JAR file and **uszipsv.csv**:

```
spark2-submit --jars spark-sql-kafka-0-10_2.11-2.3.0.jar --files uszipsv.csv driver.py
```

- The program is executed and the desired output is displayed in the console with columns **card_id**, **member_id**, **amount**, **pos_id**, **postcode**, **transaction_dt_ts** and **status**



```
-------------------------------------------
Batch: 0
-------------------------------------------
+----------------+------------+------+--------------+--------+-------------------+-------+
|card_id         |member_id   |amount|pos_id        |postcode|transaction_dt_ts  |status |
+----------------+------------+------+--------------+--------+-------------------+-------+
|348702330256514 |37495066290 |4380912|248063406800722|96774   |2017-12-31 08:24:29|GENUINE|
|348702330256514 |37495066290 |6703385|786562777140812|84758   |2017-12-31 04:15:03|FRAUD  |
|348702330256514 |37495066290 |7454328|466952571393508|93645   |2017-12-31 09:56:42|GENUINE|
|348702330256514 |37495066290 |4013428|45845320330319 |15868   |2017-12-31 05:38:54|GENUINE|
|348702330256514 |37495066290 |5495353|545499621965697|79033   |2017-12-31 21:51:54|GENUINE|
|348702330256514 |37495066290 |3966214|369266342272501|22832   |2017-12-31 03:52:51|GENUINE|
|348702330256514 |37495066290 |1753644|9475029292671  |17923   |2017-12-31 00:11:30|FRAUD  |
|348702330256514 |37495066290 |1692115|27647525195860 |55708   |2017-12-31 17:02:39|GENUINE|
|5189563368503974|1178263015 30|9222134|525701337355194|64002   |2017-12-31 20:22:10|GENUINE|
|5189563368503974|1178263015 30|4133848|182031383443115|26346   |2017-12-31 01:52:32|FRAUD  |
|5189563368503974|1178263015 30|8938921|799748246411019|76934   |2017-12-31 05:20:53|FRAUD  |
|5189563368503974|1178263015 30|1786366|131276818071265|63431   |2017-12-31 14:29:38|GENUINE|
|5189563368503974|1178263015 30|9142237|564240259678903|50635   |2017-12-31 19:37:19|GENUINE|
|5407073344486464|1147922084344|6885448|887913906711117|59031   |2017-12-31 07:53:53|FRAUD  |
|5407073344486464|1147922084344|4028209|1162660511 18182|80118   |2017-12-31 01:06:50|FRAUD  |
|5407073344486464|1147922084344|3858369|896105817613325|53820   |2017-12-31 17:37:26|GENUINE|
|5407073344486464|1147922084344|9307733|729374116016479|14898   |2017-12-31 04:50:16|FRAUD  |
|5407073344486464|1147922084344|4011296|543373367319647|44028   |2017-12-31 13:09:34|GENUINE|
|5407073344486464|1147922084344|9492531|211980095659371|49453   |2017-12-31 14:12:26|GENUINE|
|5407073344486464|1147922084344|7550074|345533088112099|15030   |2017-12-31 02:34:52|FRAUD  |
+----------------+------------+------+--------------+--------+-------------------+-------+
only showing top 20 rows
```

- In Hbase Shell, scanning **card_transactions** table, the count of rows after execution is over 59000