Scripts Execution

⇒ **For this capstone project final submission, I have submitted below artifacts:**

Task 5: Create a streaming data processing framework that ingests real-time POS transaction data from Kafka. The transaction data is then validated based on the three rules' parameters (stored in the NoSQL database) discussed previously.
Task 6: Update the transactions data along with the status (fraud/genuine) in the card_transactions table.
Task 7: Store the 'postcode' and 'transaction_dt' of the current transaction in the look-up table in the NoSQL database if the transaction was classified as genuine.

Python scripts – driver.py, rules.py, dao.py, geo_map.py

⇒ **Explanation of the solution to the streaming layer problem**

To read the data from Kafka topic following steps were taken:

- Connect to EMR
- Card_transactions and look_up_table data should be present in Hbase (*completed in mid-submission)*
- Ensure thrift server is up and running.
- Create a directory names as *python* and *src* in EMR cluster. Within *src* directory place the *driver.py* file and create 2 other directories names as *db* & *rules*.
- Create a *src.zip* file within *src* directory.

```
[[hadoop@ip-172-31-46-227 src]$ zip src.zip __init__.py rules/* db/*
  adding: __init__.py (stored 0%)
  adding: rules/__init__.py (stored 0%)
  adding: rules/rules.py (deflated 41%)
  adding: db/dao.py (deflated 61%)
  adding: db/geo_map.py (deflated 56%)
  adding: db/__init__.py (stored 0%)
[[hadoop@ip-172-31-46-227 src]$ ls
 __init__.py  db  driver.py  rules  src.zip  uszipsv.csv
```

- Run below commands to execute the kafka streaming:
  - export SPARK_KAFKA_VERSION=0.10
  - spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.1 --files uszipsv.csv driver.py

```
[[hadoop@ip-172-31-46-227 python]$ cd src/
[[hadoop@ip-172-31-46-227 src]$ export SPARK_KAFKA_VERSION=0.10
```

```
[hadoop@ip-172-31-46-227 src]$ spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5 --py-files src.zip --files uszipsv.csv driver.py
Ivy Default Cache set to: /home/hadoop/.ivy2/cache
```

- The program is executed and the desired output is displayed in the console with coulumns card_id, member_id, amount, pos_id, postcode, transaction_dt_ts and status.
- Card_transactions table in Hbase is also updated with the Genuine/Fraud records

⇒ **Stepwise approach with screenshots:**

1. **<u>driver.py script</u> –** Various user defined functions will be called within this program, which will contact given dao.py and geo_map.py scripts to call the look up table for required reasons.

- All the necessary libraries and functions have been imported.

driver.py - /Users/admin/Downloads/Ca

```
# Streaming Application to read from Kafka
# This should be the driver file for your project

# Importing required function and libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from datetime import datetime
```

- Spark Session and Spark Context is created for Credit card fraud detection application

```
# Establishing Spark Session
spark = SparkSession  \
        .builder  \
        .appName("CapStone_Project")  \
        .getOrCreate()

spark.sparkContext.setLogLevel('ERROR')

#Creating Spark context
sc = spark.sparkContext
```

- Required python files (*dao.py, geo_map.py & rules.py*) are added into the driver program

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

# Importing all the required modules
import dao
import geo_map
import rules
```

- Reading data stream from given Kafka topic.
  Bootstrap-server: 18.211.252.152
  Port Number: 9092
  Topic: transactions-topic-verified

```
# Reading data from Kafka & Topic given
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()
```

- Data frame schema is defined and data is parsed into the *df_parsed*

```
# Schema definition to read data properly
schema =  StructType([
                StructField("card_id", StringType()),
                StructField("member_id", StringType()) ,
                StructField("amount", IntegerType()),
                StructField("pos_id", StringType()),
                StructField("postcode", StringType()),
                StructField("transaction_dt", StringType())])


# Casting raw data as string and aliasing
Readable = lines.select(from_json(col("value") \
                                    .cast("string") \
                                    ,schema).alias("parsed"))

# Parsing the dataframe into df_parsed
df_parsed = Readable.select("parsed.*")
```

- UDF for calculating credit score from the look_up_table using 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()))


# Function for Credit Score
def score_data(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_data,StringType())

#Adding score column
df_parsed = df_parsed.withColumn("score",Score_udf(df_parsed.card_id))
```

- UDF for calculating postal code from the look_up_table using card_id

```
# Function for Postal Code
def postcode_data(a):
        hdao = 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_data,StringType())

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

- UDF for calculating UCL (upper control limit)

```
# Function for UCL
def ucl_data(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_data,StringType())

# Adding UCL Column
df_parsed = df_parsed.withColumn("UCL",UCL_udf(df_parsed.card_id))
```

- UDF for distance calculation between previous and current transaction postal codes from the look up tables and kafka stream (this function will use geo_map.py script)

```
#Function for Distance calculation
def distance_calc(last_postcode,postcode):
        gmap = geo_map.GEO_Map.get_instance()
        last_lat = gmap.get_lat(last_postcode)
        last_lon = gmap.get_long(last_postcode)
        lat = gmap.get_lat(postcode)
        lon = gmap.get_long(postcode)
        final_dist = gmap.distance(last_lat.values[0],last_lon.values[0],lat.values[0],lon.values[0])
        return final_dist

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

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

- UDF for date and time calculation. Further time_diff is calculated using last transaction date and current transaction date. Time_diff acts as one of the rules in identifying genuine/fraud transactions.

```
# Function for Time calculation
def time_cal(last_date, curr_date):
        diff= curr_date-last_date
        return (diff.total_seconds())/3600

# Function for Transaction Date
def lTransD_data(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
lTransD_udf = udf(lTransD_data,StringType())

# Adding Transaction Date Column
df_parsed = df_parsed.withColumn("last_transaction_date",lTransD_udf(df_parsed.card_id))


# Defining UDF for Calculating Time
time_udf = udf(time_cal,DoubleType())

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

# Adding Time diff 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_find(card_id,member_id,amount,pos_id,postcode,transaction_dt,transaction_dt_ts,last_transaction_date_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=card_id,table='look_up_table')
        status = 'FRAUD'
        if rules.rules_check(data_fetch['info:UCL'],score,speed,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())
```

- Selecting the required columns from the parsed streaming data frame and writing output to the console.

```
# 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.time_diff))


# Displaying only required columns
df_parsed = df_parsed.select("card_id","member_id","amount","pos_id","postcode","transaction_dt_ts","status")


# Printing Output on Console
query1 = df_parsed \
        .writeStream  \
        .outputMode("append")  \
        .format("console")  \
        .option("truncate", "False")  \
        .start()

# query termination command
query1.awaitTermination()
```

2. **<u>rules.py script</u>** – This program will check whether incoming transaction is a genuine or fraud. Follwing rules are defined in the script:
    1. Transaction amount < UCL *(Amount of transaction should be less than UCL)*
    2. Time difference in sec < 4 times the distance *(Time difference between current transaction timestamp and last transaction timestamp)*
    3. Credit Score < 200

```
rules.py - /Users/admin/Downloads/Capstone_Project/
# List all the functions to check for the rules

# Function 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
```

3. <u>Console Output</u>

```
-------------------------------------------
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|117826301530|9222134|525701337355194|64002   |2017-12-31 20:22:10|GENUINE|
|5189563368503974|117826301530|4133848|182031383443115|26346   |2017-12-31 01:52:32|FRAUD  |
|5189563368503974|117826301530|8938921|799748246411019|76934   |2017-12-31 05:20:53|FRAUD  |
|5189563368503974|117826301530|1786366|131276818071265|63431   |2017-12-31 14:29:38|GENUINE|
|5189563368503974|117826301530|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|116266051118182|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
```

4. Navigate to Hbase Shell, check *card_transactions* table
   scan ‘card_transactions)

   Total count of rows in *card_transactions* table in HBase is 59367

```
Current count: 20000, row: 27999
Current count: 21000, row: 28899
Current count: 22000, row: 29799
Current count: 23000, row: 30698
Current count: 24000, row: 31598
Current count: 25000, row: 32498
Current count: 26000, row: 33398
Current count: 27000, row: 341724964458347.210778177559185.12-06-2018152638.2021-01-04171328.398477
Current count: 28000, row: 346618652451637.540752175696215.29-04-2018005259.2021-01-04171400.227023
Current count: 29000, row: 35264
Current count: 30000, row: 36164
Current count: 31000, row: 370582035866789.433646648625434.08-07-2018034337.2021-01-04171349.489639
Current count: 32000, row: 375806375521605.880937166605469.26-05-2018130045.2021-01-04171430.733012
Current count: 33000, row: 38176
Current count: 34000, row: 39076
Current count: 35000, row: 39977
Current count: 36000, row: 40768
Current count: 37000, row: 41560
Current count: 38000, row: 42387
Current count: 39000, row: 4318541450654035.496612742732167.12-02-2018145807.2021-01-04171356.009418
Current count: 40000, row: 43999
Current count: 41000, row: 44784
Current count: 42000, row: 45546
Current count: 43000, row: 46306
Current count: 44000, row: 47134
Current count: 45000, row: 47925
Current count: 46000, row: 48730
Current count: 47000, row: 49500
Current count: 48000, row: 50351
Current count: 49000, row: 5120
Current count: 50000, row: 51888
Current count: 51000, row: 5257502990314019.205172644364018.14-07-2018070014.2021-01-04171327.867742
Current count: 52000, row: 53290
Current count: 53000, row: 5620
Current count: 54000, row: 6211
Current count: 55000, row: 6478888441720966.273246841077378.06-10-2018212851.2021-01-04171333.585477
Current count: 56000, row: 6968
Current count: 57000, row: 7868
Current count: 58000, row: 8768
Current count: 59000, row: 9668
59367 row(s) in 3.8140 seconds

=> 59367
hbase(main):003:0>
```