## **Scripts Execution**

**Prerequisites:** All the steps for static part of this project are completed and we have two tables in HBase as given below:

1. **card\_transactions: This table contains all historical transactions along with its status as GENUINE or FRAUD**
2. **lookup\_hbase: This table contains lookup data such as UCL, date and postcode of last transaction, member id, score for all 999 card id’s.**

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

Code for streaming application is written to process the incoming steaming data from Kafka and categorize the transaction as GENUINE or FRAUD based on three rules.

Whole application contains various files as given below:

1. driver.py: This python file act as a driver file of program which reads streaming data from Kafka and process it to categorize incoming transaction as a GENUINE OR FRAUD with the help of other files.

1. rules.py: This python file contains the three functions which are being called from driver.py file to check the business rules. These rules together will help to determine transaction as GENUINE or FRAUD.
2. dao.py: This file contains code for operations regarding database which will help to read and write data from HBase tables as per requirements.
3. geo\_map.py: This file contains functions regarding location details and will be helpful to calculate distance between two places based on post id.

**Code Explanation:**

1. Import all required modules and packages

from pyspark.sql import SparkSession  
from pyspark.sql.functions import \*  
from pyspark.sql.types import \*  
import uuid

1. Start the spark session with application name as spark-streaming

*# initialising Spark session*spark = SparkSession \  
 .builder \  
 .appName(**"spark-streaming"**) \  
 .getOrCreate()  
spark.sparkContext.setLogLevel(**'ERROR'**)

1. Adding and importing all required files (dependencies) to spark context so that functions from those files can be used in our code

*# adding required python files*spark.sparkContext.addPyFile(**'db/dao.py'**)  
spark.sparkContext.addPyFile(**'db/geo\_map.py'**)  
spark.sparkContext.addPyFile(**'rules/rules.py'**)

*# importing modules from python files*from dao import \*  
from geo\_map import \*  
from rules import \*

1. Custom JSON Schema is created to map it with incoming data from Kafka Server

*# defining schema for a single order*jsonSchema = StructType() \  
 .add(**"card\_id"**, StringType()) \  
 .add(**"member\_id"**, StringType()) \  
 .add(**"amount"**, LongType()) \  
 .add(**"postcode"**, StringType()) \  
 .add(**"pos\_id"**, StringType()) \  
 .add(**"transaction\_dt"**, StringType())

1. Read the stream coming from Kafka by providing server and topic details and then map this raw data to custom schema created in last step

*# reading input from Kafka*orderRawData = spark.readStream \  
 .format(**"kafka"**) \  
 .option(**"kafka.bootstrap.servers"**, **"18.211.252.152:9092"**) \  
 .option(**"startingOffsets"**, **"earliest"**) \  
 .option(**"failOnDataLoss"**, **"false"**) \  
 .option(**"subscribe"**, **"transactions-topic-verified"**) \  
 .load()  
  
*# creating an order stream for reading data from json in kafka*orderStream = orderRawData.select(from\_json(col(**"value"**).cast(**"string"**), jsonSchema).alias(**"data"**)).select(**"data.\*"**)

1. Creating functions to process this stream

6.1 getData function -

*# Define function to get data from hbase lookup table for given card\_id*def getData(card\_id):  
 lookupdata = HBaseDao.get\_instance().get\_data(card\_id, **'lookup\_hbase'**)  
 dic = {}  
 for x in lookupdata:  
 dic[x.decode()] = lookupdata[x].decode()  
 return dic

This function will take card id as an input parameter. HBaseDao.get\_instance() makes a connection with HBase and gives an instance of the class. On that instance we are calling get\_data function which have two input parameters as card id and name of the table. Here we are passing ‘lookup\_hbase’ as table name. This function will give the data present in ‘lookup\_hbase’ table for given card id.

Type of this data being returned is in the form of dictionary which is byte encoded, hence it is converted in simple dictionary by decoding both of its key and value and returned from function.

* 1. distance function -

*# Define function to get distance between two postcodes of recent transaction*def distance(currentPostcode, lookupdata):  
 oldPostcode = lookupdata[**'lookup\_column\_family:postcode'**]  
 return GEO\_Map.get\_instance().distance(GEO\_Map.get\_instance().get\_lat(currentPostcode),GEO\_Map.get\_instance().get\_long(currentPostcode),GEO\_Map.get\_instance().get\_lat(oldPostcode),GEO\_Map.get\_instance().get\_long(oldPostcode))

This function will take postcode of current transaction being processed and lookupdata which was returned by getData function as an input parameters. From the lookupdata we are extracting the value of postcode of last transaction.

GEO\_Map.get\_instance() gives an instance of GEO\_Map calss which has its own distance method. This distance methods takes the latitude and longitude as input parameters and gives the distance in KM. This latitude and longitude can be determined by calling get\_long and get\_lat functions of the same GEO\_Map calss by passing postcodes.

So finally, this function will return the distance between two postcodes in KM where current and previous transaction were done.

* 1. determineTransaction function -

*# Define function to determine status of transaction*def determineTransaction(current\_transaction\_date, distance, amount, lookupdata):  
 if (Rules.get\_instance().checkScore(lookupdata[**'lookup\_column\_family:score'**]) and Rules.get\_instance().checkUcl(amount, lookupdata[**'lookup\_column\_family:ucl'**]) and Rules.get\_instance().checkSpeed(lookupdata[**'lookup\_column\_family:transaction\_dt'**], current\_transaction\_date, distance)):  
 return **"GENUINE"** else:  
 return **"FRAUD"**

This function will take current\_transaction\_date, distance, amount and lookupdata as input parameters.

This function will again try to call other three functions from Rules class which is present in rules.py file. These three functions are checkScore, checkUcl and checkSpeed which checks the business rules. If all of these functions returns True then this transaction will be considered as GENUINE otherwise as FRAUD transaction and will return a string either as GENUINE or FRAUD.

* + 1. checkScore function

def checkScore(self, score):  
 if int(score) < 200:  
 return False  
 else:  
 return True

This function returns True if score is greater than 200.

* + 1. checkUcl function

def checkUcl(self, amount, ucl):  
 if amount > int(ucl):  
 return False  
 else:  
 return True

This function returns False if amount of transaction is greater than UCL, otherwise returns False.

* + 1. checkSpeed function

def checkSpeed(self, last\_transaction\_date, current\_transaction\_date, distance):  
 curts = datetime.strptime(current\_transaction\_date, **'%d-%m-%Y %H:%M:%S'**)  
 lasts = datetime.strptime(last\_transaction\_date, **'%d-%m-%Y %H:%M:%S'**)  
 timeinterval = (curts - lasts).total\_seconds()  
 speed = distance / timeinterval  
 if (speed > 0.25):  
 return False  
 else:  
 return True

This function calculates speed based on the transaction date of current and last transaction and distance which are passed as parameters to this function. If the speed is greater than 0.25 Km/s then returns False else returns True.

* 1. dataForTransaction function –

*# Define function to add transaction record to card\_transactions table*def dataForTransaction(card\_id, member\_id, amount, postcode, pos\_id, transaction\_dt, status):  
 row = {}  
 key1 = **"ctf:card\_id"** key2 = **"ctf:member\_id"** key3 = **"ctf:amount"** key4 = **"ctf:postcode"** key5 = **"ctf:pos\_id"** key6 = **"ctf:transaction\_dt"** key7 = **"ctf:status"** row[key1.encode()] = str(card\_id).encode()  
 row[key2.encode()] = str(member\_id).encode()  
 row[key3.encode()] = str(amount).encode()  
 row[key4.encode()] = str(postcode).encode()  
 row[key5.encode()] = str(pos\_id).encode()  
 row[key6.encode()] = str(transaction\_dt).encode()  
 row[key7.encode()] = str(status).encode()  
 HBaseDao.get\_instance().write\_data(str(uuid.uuid1()).encode(), row, **'card\_transactions'**)  
 return **"OK"**

Thisfunction takes all parameters such as card\_id, member\_id, amount, postcode, pos\_id, transaction\_dt and status which needs to write in 'card\_transactions’ table in HBase. The parameters being passed to this function are in simple form such as string or long type but to write it into HBase it needs to be convert in byte form. If write to the HBase is successful then this function will return OK as string.

* 1. dataForLookup function –

*# Define function to add genuine transaction record to lookup table*def dataForLookup(key, current\_transaction\_date, postcode):  
 row = {}  
 key1 = **"lookup\_column\_family:transaction\_dt"** key2 = **"lookup\_column\_family:postcode"** row[key1.encode()] = str(current\_transaction\_date).encode()  
 row[key2.encode()] = str(postcode).encode()  
 HBaseDao.get\_instance().write\_data(key.encode(), row, **'lookup\_hbase'**)  
 return **"OK"**

Thisfunction takes all parameters such as key which will be card id, current\_transaction\_date and postcode which needs to update in 'lookup\_hbase’ table in HBase. Similar to previous function parameters are converted in byte form. If write to the HBase is successful then this function will return OK as string.

1. All of these functions can not be used directly in streaming application hence those are converted into UDFs

*# creating UDFs of functions to use in pyspark*getDataUDF = udf(lambda x : getData(x), MapType(StringType(), StringType()))  
distanceUDF = udf(lambda x,y : distance(x,y), FloatType())  
determineTransactionUDF = udf(lambda x,y,z,w : determineTransaction(x,y,z,w))  
dataForLookupUDF = udf(lambda x,y,z : dataForLookup(x,y,z))  
dataForTransactionUDF = udf(lambda x,y,z,w,l,m,n : dataForTransaction(x,y,z,w,l,m,n))

1. Calling all UDFs one by one which eventually will call functions and process the data and categorize it as GENUINE OR FRAUD. Here df4 data will have the result in ‘flag’ column as GENUINE or FRAUD. This data will be written to the ‘card\_transactions’ table when dataForTransactionUDF is called on it.

Same df4 streaming data frame is filtered where flag is ‘GENUINE’ and then UDF dataForLookupUDF is called over it so that it will just update the ‘lookup\_hbase’ table only for Genuine transactions.

df2 = orderStream.withColumn(**"lookupData"**, getDataUDF(orderStream.card\_id))  
df3 = df2.withColumn(**"distance"**, distanceUDF(df2.postcode, df2.lookupData))  
df4 = df3.withColumn(**"flag"**, determineTransactionUDF(df3.transaction\_dt, df3.distance, df3.amount, df3.lookupData))

df5 = df4.withColumn(**"transactionResult"**, dataForTransactionUDF(df4.card\_id, df4.member\_id, df4.amount,df4.postcode, df4.pos\_id, df4.transaction\_dt, df4.flag))

df6 = df4.filter(col(**"flag"**) == **"GENUINE"**).withColumn(**"lookupResult"**, dataForLookupUDF(df4.card\_id, df4.transaction\_dt, df4.postcode))

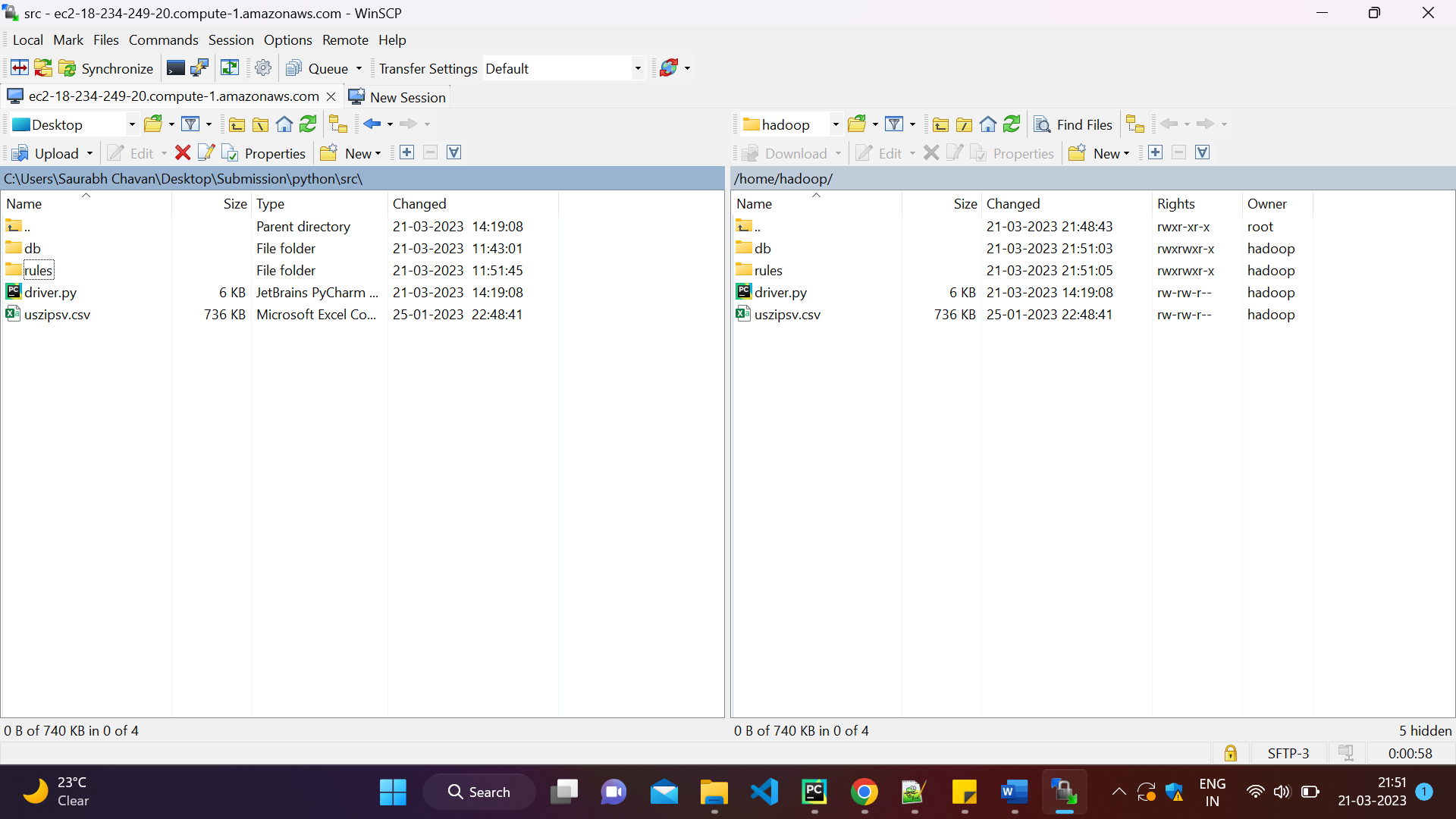
1. Till this point we just defined functions, UDF’s and called those on streaming data frame for transformations. But this will just create the execution plan and no actual processing would be done still we call some action. Hence We are starting these three streams of which result of two streams will be printed in file storage and for one stream it will print result on console.

query0 = df5 \  
 .select(**"card\_id"**,**"member\_id"**,**"amount"**,**"pos\_id"**,**"postcode"**,**"transaction\_dt"**,**"flag"**) \  
 .writeStream \  
 .outputMode(**"append"**) \  
 .format(**"console"**) \  
 .option(**"truncate"**, **"false"**) \  
 .trigger(processingTime=**"1 minute"**) \  
 .start()  
  
  
query1 = df5 \  
 .select(**"\*"**) \  
 .writeStream \  
 .outputMode(**"append"**) \  
 .format(**"json"**) \  
 .option(**"format"**, **"append"**) \  
 .option(**"truncate"**, **"false"**) \  
 .option(**"path"**, **"card\_transactions\_records"**) \  
 .option(**"checkpointLocation"**, **"card\_transactions\_records\_checkpoint"**) \  
 .option(**"truncate"**, **"False"**) \  
 .trigger(processingTime=**"1 minute"**) \  
 .start()  
  
  
  
query2 = df6 \  
 .select(**"\*"**) \  
 .writeStream \  
 .outputMode(**"append"**) \  
 .format(**"json"**) \  
 .option(**"format"**, **"append"**) \  
 .option(**"truncate"**, **"false"**) \  
 .option(**"path"**, **"lookup\_hbase\_records"**) \  
 .option(**"checkpointLocation"**, **"lookup\_hbase\_records\_checkpoint"**) \  
 .option(**"truncate"**, **"False"**) \  
 .trigger(processingTime=**"1 minute"**) \  
 .start()

query0.awaitTermination()  
query1.awaitTermination()  
query2.awaitTermination()

**The steps to run the code:**

1. Upload all files to hadoop by using WinSCP



1. Install happybase and pandas

**sudo yum update**

**yum install gcc**

**sudo yum install python3-devel**

**pip install happybase**

**pip install pandas**

1. Run code by using spark-submit command

**spark-submit --packages org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.5 driver.py**

Once job is submitted by using above command it is expected that the code will process incoming streaming data from Kafka by calling functions and UDFs and will classify the transaction as GENUINE or FRAUD. This action will add the all new records to ‘card\_transactions’ table and update ‘lookup\_hbase’ table for GENUINE transactions.

**NOTE: Screen Shots are not attached as there is problem with Kafka server and no data being streamed from Kafka topic.**