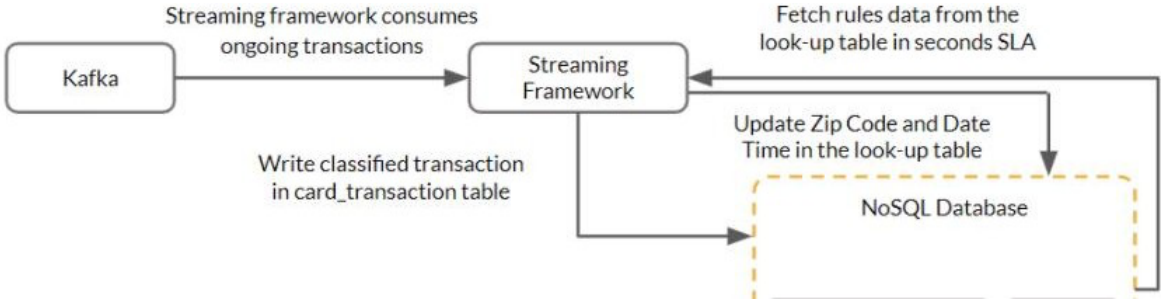
Scripts Execution

Explanation of the solution to the streaming layer problem

In today's world Credit Card Fraud is a common problem. For the solution to this problem there is a feature to detect fraudulent transactions, wherein once card member swipes his/her card for payment the transaction should be classified as fraudulent or authentic based on the Predefined rules. If Fraud is detected transaction must be declined.

Here in this part of project we are focused on the Streaming Layer Solution



Below are Tasks Performed here:

1. Created a streaming framework that ingests real-time POS transaction data from kafka.
2. Data is validated based on the rules for UCL, score and speed
3. Update the transaction data with status as "FRAUD" or "GENUINE" in card_transactions table
4. If the data is "GENUINE", update the transaction_code and postcode in the lookup table.

Driver Program is the main program which contains the complete logic. This program will call the DB files to connect to Hbase tables and calculating distance and also will call rules.py for calculating the rules based on which any transaction is declared "FRAUD" or "GENUINE"

Steps for running the code

1. Connect to putty and login as ec2-user
2. Switch to root user (sudo -i)
3. Start Thrift server

/opt/cloudera/parcels/CDH/lib/hbase/bin/hbase-daemon.sh start thrift -p 9090

4. **Connect to hbase using hbase shell command and check the tables created. This table will be used in driver code.**

5. **Create a directory python and place the src files under that.**

The directory structure consists of below:

db files:

1. **dao.py** – This file connects to HBase and access the tables in Hbase using driver.py
   This file connects using a host which we need to specify.
   We need to update self.host to localhost for running the program
2. **geo_map.py** – This file contains code to find the distance between two postcodes and fetches data from uszipsv.csv file

rules files:

1. rules.py – This file fetch the rules from lookup_table

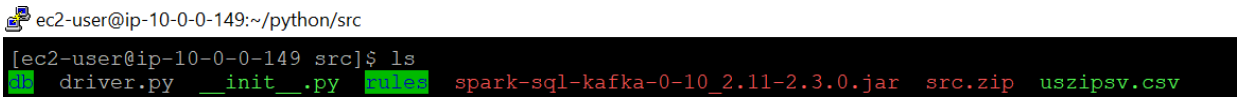
driver.py:

This is the main program which calls the db files and rule file and contains all necessary logic for updating the card_transactions and lookup_table

Jar file:

We will download the necessary jar file for running spark streaming
wget https://ds-spark-sql-kafka-jar.s3.amazonaws.com/spark-sql-kafka-0-10_2.11-2.3.0.jar

uszipsv.csv file is the file being called in geo_map.py for distance calculation.



6. Run the code

Create zip file for src files

zip src.zip__init__.py rules/* db/*

For running code need to Export Spark kafka version
export SPARK_KAFKA_VERSION=0.10
Run below command for spark2-submit and mention the files
spark2-submit --py-files src.zip --jars spark-sql-kafka-0-10_2.11-2.3.0.jar --files uszipsv.csv driver.py

All the necessary logic will run and update the card_transactions table in Hbase.

```
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
```

```
-------------------------------------------
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|7865627771408l2|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|3692663422725O1|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|5257013373551941|64002   |2017-12-31 20:22:10|GENUINE|
|5189563368503974|117826301530 |4133848|18203138344311S|26346   |2017-12-31 01:52:32|FRAUD  |
|5189563368503974|117826301530 |8938921|79974824641101I|76934   |2017-12-31 05:20:53|FRAUD  |
|5189563368503974|117826301530 |1786366|131276818071265|63431   |2017-12-31 14:29:38|GENUINE|
|5189563368503974|117826301530 |9142237|5642402596789O3|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|5433733673196471|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  |
+----------------+-------------+-------+---------------+--------+-------------------+-------+
```

Logic of Main Program (driver.py)

1. **Import the libraries and define system variables**

2. **Import the db and rules files**

3. **Call the Spark Streaming for reading the Streaming data using readStream from kafka**

Topic: transactions-topic-verified
Kafka bootstrap servers: 18.211.252.152:9092

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

4. **Define the Schema definition and parse data**

```
# Schema definition
MySchema =  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 read from Kafka topic
data=lines.select(from_json(col("value").cast("string"),MySchema).alias("parsed"))
```

5. **UDF's is defined for fetching the data**

Fetch_score_data:

get_instance and get_data is called from dao.py. It takes card_id as parameter and returns score.
Using withColumn added the score column to the txn_df

```
def fetch_score_data(card_id):
    hdao = dao.HBaseDao.get_instance()
    data_fetch = hdao.get_data(key=card_id,table='lookup_table')

    return data_fetch['info:score']

#Score_udf is defined
Score_udf = udf(fetch_score_data,StringType())

#adding the column score to the dataframe using withColumn
txn_df = txn_df.withColumn("score",Score_udf(txn_df.card_id))
```

Fetch_postcode_data

get_instance and get_data is called from dao.py. It takes card_id as parameter and returns postcode.
Using withColumn added the last_postcode column to the txn_df

```
def fetch_postcode_data(card_id):
    hdao = dao.HBaseDao.get_instance()
    data_fetch = hdao.get_data(key=card_id,table='lookup_table')

    return data_fetch['info:postcode']

#poscode_udf is defined
postcode_udf = udf(fetch_postcode_data,StringType())

#adding the column last_postcode using withColumn to the dataframe
txn_df = txn_df.withColumn("last_postcode",postcode_udf(txn_df.card_id))
```

Fetch_lTransD_data:
get_instance and get_data is called from dao.py. It takes card_id as parameter and returns last transaction date.
Using withColumn added the last_transaction_date and last_transaction_date_ts (timestamp) column to the txn_df

```
def fetch_lTransD_data(card_id):
    hdao = dao.HBaseDao.get_instance()

    data_fetch = hdao.get_data(key=card_id,table='lookup_table')

    return data_fetch['info:transaction_date']

#lTransD udf is defined
lTransD_udf = udf(fetch_lTransD_data,StringType())

#adding the column last_transaction_date using withColumn to the dataframe
txn_df = txn_df.withColumn("last_transaction_date",lTransD_udf(txn_df.card_id))
```

Fetch_ucl_data:

get_instance and get_data is called from dao.py. It takes card_id as parameter and returns Upper card limit (UCL)
Using withColumn added the UCL column to the txn_df

```
def fetch_ucl_data(card_id):
    hdao = dao.HBaseDao.get_instance()

    data_fetch = hdao.get_data(key=card_id,table='lookup_table')

    return data_fetch['info:UCL']

#UCL_udf is defined
UCL_udf = udf(fetch_ucl_data,StringType())

#added column UCL using withColumn
txn_df = txn_df.withColumn("UCL",UCL_udf(txn_df.card_id))
```

Calc_distance:
distance calculation is done by calling GEO_Map.get_instance(),get latitude,longitude from the methods provided in the db file

Added column distance to txn_df dataframe

```
def calc_distance(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

#distance_udf is defined
distance_udf = udf(calc_distance,DoubleType())

#added column "distance"
txn_df = txn_df.withColumn("distance",distance_udf(txn_df.last_postcode,txn_df.postcode))
```

Calc_time:

This is calculated as curr_date-last_date

```
def cal_time(last_date, curr_date):


    diff= curr_date-last_date
    return (diff.total_seconds())/3600
```

Calc_speed : Using distance and time speed is calculated

```
#calc_speed function is defined
#speed= distance/time
def calc_speed(dist,time):
    if time<0:
        time=time*(-1)
    if time==0:
        return -1.0
    return (dist)/time

speed_udf = udf(calc_speed, DoubleType())
```

Find_status:

This function will take all the above calculated columns as input and apply the rules and check if the transaction is “FRAUD” or “GENUINE”

Below code will also update the card_transactions table with status and lookup_table if the status is “GENUINE”

Hdao.write_date is used for writing data to the card_transactions and lookup_table in the Hbase

Rules.py

This will take UCL, score, speed and amount as parameter
And check the code as below: (Pseudocode)
If amount < UCL (upper card limit)
  If score >200
    If speed greater than 0 and less than 1000
      Then it will return True i.e “GENUINE” else False i.e “FRAUD”

```
def find_status(card_id,member_id,amount,pos_id,postcode,transaction_dt,transaction_dt_ts,last_transacti

        hdao = dao.HBaseDao.get_instance()
        geo = geo_map.GEO_Map.get_instance()
        look_up = hdao.get_data(key=card_id,table='Lookup_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,'Lookup_table')


        row = {'info:postcode':bytes(postcode),'info:pos_id':bytes(pos_id),'info:card_id':bytes(card_id)
        key = '{0}.{1}.{2}.{3}'.format(card_id,member_id,str(transaction_dt),str(datetime.now())).replac
        hdao.write_data(bytes(key),row,'card_transactions')
        return status

status_udf = udf(find_status,StringType())
txn_df = txn_df.withColumn('status',status_udf(txn_df.card_id,txn_df.member_id,txn_df.amount,txn_df.pos_

final_df = txn_df.select("card_id","member_id","amount","pos_id","postcode","transaction_dt_ts","status"
```

Write dataframe to console using writestream

```
CreditCardTxn= final_df \
               .writeStream  \
               .outputMode("append")  \
               .format("console")  \
               .option("truncate", "False")  \
               .trigger(processingTime="1 minute") \
               .start()

CreditCardTxn.awaitTermination()
```