Scripts Execution
Explanation of the solution to the batch layer problem

- In this project our goal is to detect the credit card fraud in
  the real time streaming transaction.

- This whole project is divided into two part. In first part of
  document we will focus on first half of the project and in the second part of document we will focus on rest of the project.

First half of the project:

- Task 1: Load the transactions history data (card_transactions.csv) in a NoSQL database.

- Task 2: Ingest the relevant data from AWS RDS to Hadoop.

- Task 3: Create a look-up table with columns specified earlier in the problem statement.

- Task 4: After creating the table, you need to load the relevant data in the lookup table.
  • In order to perform the aforementioned tasks, I have created a single node EMR cluster with the below configuration.
  • **TASK 1:**

    - To load the data from csv file to Hbase (NoSQL database). I have uploaded the csv file to S3 bucket. Loc: s3://capstone-aditya/input/card_transactions.csv

    - Then from S3, I have imported the file into hdfs using distcp command.

- After that I have opened the hive shell and created a table to hold the data from csv file and then created hive-hbase integrated table so that all data must be reflected in

Hbase also.

- After importing data to hive-hbase integrated table, I used random UUID for row key creation.

- Finally verified all the records imported successfully into table.

- **TASK 2:**
  - In order to ingest data from AWS RDS to hdfs, I have
    used sqoop to import the data.

  - After importing the data into hdfs, I used hive to create
    and load the imported data into hive table.

  - Finally verified all the records imported successfully into
    table.

• **TASK 3:**
• In order to create Lookup table I have used hive-base
integrated table.
• **TASK 4:**

- To load the data into Lookup table first we need to
calculate some fields.

- I have created two intermediate table in hive, first one to
hold the last 10 transactions and second one to
calculated the UCL of last 10 transactions.

- Finally load the data into lookup table.

- Finally verified all the records imported successfully into
table.
• This is the logic behind my way to

complete till task 4.

Second half of the project:

- **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.**
  • List of documents submitted in the zip:
  • run.py - Task 5, Task6 & Task7

• LogicFinal.pdf - explaining the logic and steps followed

- **TASK5:**
  - In order to process real time data, I have used kafka streaming and written a python script. The script will read the data stream and create a pyspark data frame.

  - The next thing is to check all the three condition to validate a transaction. In order to do so I have joined the streaming data frame with the lookup table (NoSQL Database) on card id so that we have one single data frame with all the streaming data info along with info related to UCL, Score and last Transaction Date.

  - After that I have read the uszip.csv in data frame and join it with the original data frame on postcode.

- The final data frame have the streaming data, data from lookup table (ucl, score and transaction date) and data related to latitude and longitude (of both the streaming data and the lookup table data).

- Finally I compare different columns (amount with ucl, score greater than 200 or not and based to latitude and longitude calculated the distance travelled and compare it against time which is calculated from difference between streaming transaction date and last transaction date) of each row to classify whether transaction is fraud or genuine.

- Based on all the condition a new column ‘STATUS’ is added in the data frame with value either ‘Genuine’ or ‘Fraud’ and all the unnecessary columns have been dropped.

- **TASK6:**

• The final data frame will have the info of streaming data along with status as Genuine or Fraud.

• This data frame finally merged to the card transaction table in HBase.

• **TASK7:**

• All those transaction which is classified as genuine need to be added in lookup table along with postcode and transaction date info.

Script Execution (run.py)

Exporting correct kafka version: export SPARK_KAFKA_VERSION=0.10 Creating script file: vi run.py

Spark submit command: spark-submit --packages org.apache.spark:spark-sql- kafka-0-10_2.11:2.4.5 run.py

Script File:

```
# Importing Libraries
from pyspark.sql import SparkSession from pyspark.sql.functions import * from pyspark.sql.types import *
import ast
import functools

# Creation of Spark Session spark = SparkSession \

.builder \
.appName("Structured Streaming ")\ .enableHiveSupport() \ .getOrCreate()
```

```
spark.sparkContext.setLogLevel("ERROR")

# Construct a streaming DataFrame that reads from topic df =
spark \

.readStream\
.format("kafka")
\ .option("kafka.bootstrap.servers","18.211.252.152:9092")
\ .option("subscribePattern","transactions-topic-verified.*")
\ .option("startingOffsets", "earliest") \
.load()

# Take set of SQL expressions in a string to execute df1 =
df.selectExpr("CAST(value AS STRING)")



# Functions to get data from kafka in appropriate fields def
card_read(a):

b = ast.literal_eval(a)

return b['card_id'] def member_read(a):

b = ast.literal_eval(a)

return b['member_id'] def amount_read(a):

b = ast.literal_eval(a)

return b['amount'] def postcode_read(a):

b = ast.literal_eval(a)

return b['postcode'] def pos_read(a):

b = ast.literal_eval(a)

return b['pos_id'] def txn_read(a):

b = ast.literal_eval(a) return b['transaction_dt']
```

```
# Defining UDFs for putting datas in particular fields cardRead
= udf(lambda a: card_read(a),StringType()) df2 =
df1.withColumn('card_id',cardRead(df1.value))

memRead = udf(lambda a: member_read(a),StringType()) df3
= df2.withColumn('member_id', memRead(df2.value))

amtRead = udf(lambda a: amount_read(a),StringType()) df4 =
df3.withColumn('amount', amtRead(df3.value))

postRead = udf(lambda a: postcode_read(a),StringType()) df5
= df4.withColumn('postcode', postRead(df4.value))

posRead = udf(lambda a: pos_read(a),StringType()) df6 =
df5.withColumn('pos_id', posRead(df5.value))

txnRead = udf(lambda a: txn_read(a),StringType())



df7 = df6.withColumn('transaction_dt', txnRead(df6.value))

# Dropping the value column as the value is extracted from
value df8 = df7.drop('value')

# Typecasting amount to Double Type
df8.withColumn('amount',df8.amount.cast(DoubleType()))

# Reading the uszipsv csv and creating a dataframe for the
same
df_zip = spark.read.csv('s3://cardtransac/uszipsv.csv')
df_zip =
df_zip.withColumnRenamed('_c0','postcode').withColumnRena
med('_c1','lat').wit hColumnRenamed('_c2','long')

# Typecasting latitide and longitude to DoubleType
df_zip =
df_zip.withColumn('lat',df_zip.lat.cast(DoubleType())).withColu
mn('long',df_zip.lo ng.cast(DoubleType()))
```

```
# Connecting to hive database
spark.sql('use capstone_project')
df9 = spark.sql("select card_id,ucl,score from
lookup_data_hbase_orc")

# Joining the kafka stream with lookup data to get the UCL
and the Credit Score df10=df8.join(df9,'card_id','left_outer')

# Joining the updated joined dataset to get the lattitude and
longitude of Postcode
df14=df10.join(df_zip,'postcode','left_outer')

# Dropping irrelevant columns df14 = df14.drop('_c3')
df14 = df14.drop('_c4')

# Renaming Columns
df14 =
df14.withColumnRenamed('transaction_dt','transaction_dt_cur
rent').withColumnR



enamed('lat','lat_current').withColumnRenamed('long','long_cu
rrent').withColumn Renamed('postcode','postcode_current')

# Reading data from hive
df15 = spark.sql("select card_id,postcode,transaction_dt from
lookup_data_hbase_orc")

#Joining dataframes to get another set of lattitude and
longitude for the Postcode for last transaction happened
present in lookup
df16 = df14.join(df15,'card_id','left_outer')
df17 = df16.join(df_zip,'postcode','left_outer')

# Dropping irrelevant columns df17 = df17.drop('_c3')
df17 = df17.drop('_c4')
```

```
# Renaming Columns
df17 =
df17.withColumnRenamed('transaction_dt','transaction_dt_loo
kup').withColumnR
enamed('lat','lat_lookup').withColumnRenamed('long','long_loo
kup').withColumnR enamed('postcode','postcode_lookup')

# Typecasting Current Transaction column to TimeStamp type
df17 =
df17.withColumn('transaction_dt_current',unix_timestamp(df17
.transaction_dt_cu rrent,'dd-MM-yyyy
HH:mm:ss').cast("timestamp"))

df17 =
df17.withColumn('transaction_dt_lookup',unix_timestamp(df17.
transaction_dt_loo kup,'dd-MM-yyyy
HH:mm:ss').cast("timestamp"))
import math

# Defining function for calculation of distance between two
points def dist(lat1, lon1, lat2, lon2):

lat1 = math.radians(lat1) lon1 = math.radians(lon1) lat2 =
math.radians(lat2) lon2 = math.radians(lon2)



dlon = lon2 - lon1
dlat = lat2 - lat1
a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) *
math.sin(dlon / 2)**2 c = 2*math.asin(math.sqrt(a))
r = 6371

return (c*r)

# Applying UDF in distance column
distance = udf(dist, DoubleType())
```

```
df17 =
df17.withColumn('Distance',distance('lat_current','long_current'
,'lat_lookup','long_ lookup'))

# Calculation of difference between current and reference
timestamp
df18 =
df17.withColumn('diffInSeconds',col('transaction_dt_current').c
ast(DoubleType())-
col('transaction_dt_lookup').cast(DoubleType()))

df18 =
df18.withColumn('diffInHours',round(col('diffInSeconds')/3600))
df18 = df18.withColumn('Speed',col('Distance')/
col('diffInHours'))

# Classifying Fraud and Genuine
union_df = df18.withColumn("status",when((col('amount')<
col('ucl'))& (col('score') > 200) & (col('Speed') <=
900.0),'Genuine')

.when((col('amount')>=col('ucl'))| (col('score') <= 200) |
(col('Speed') > 900.0),'Fraud'))

# Dropping irrelevant columns
card_txn = union_df.drop('ucl')
card_txn = card_txn.drop('score')
card_txn = card_txn.drop('diffInSeconds') card_txn =
card_txn.drop('lat_current')
card_txn = card_txn.drop('long_current')
card_txn = card_txn.drop('lat_lookup')
card_txn = card_txn.drop('long_lookup')
card_txn = card_txn.drop('transaction_dt_lookup')
```

card_txn = card_txn.drop('postcode_lookup') card_txn = card_txn.drop('Distance') card_txn = card_txn.drop('diffInHours')

Renaming the columns and making the data rearranged in proper format card_txn = card_txn.withColumnRenamed('transaction_dt_current','transa ction_dt').withColu mnRenamed('postcode_current','postcode')

card_txn = card_txn.select('card_id','member_id','amount','postcode','pos_ id','transaction_dt', 'status')

Query to show data in the console query = card_txn\

.writeStream\ .outputMode("append")\ .option("truncate", "false") \ .format("console")\

.start()

Query Terminiation query.awaitTermination()

```
-------------------------------------------
Batch: 0
-------------------------------------------
+----------------+-------------+-------+--------+--------------+-------------------+-------+
|card_id         |member_id    |amount |postcode|pos_id        |transaction_dt     |status |
+----------------+-------------+-------+--------+--------------+-------------------+-------+
|348702330256514 |37495066290  |4380912|96774   |248063406800722|2018-03-01 08:24:29|Genuine|
|348702330256514 |37495066290  |6703385|84758   |786562777140812|2018-06-02 04:15:03|Genuine|
|348702330256514 |37495066290  |7454328|93645   |466952571393508|2018-02-12 09:56:42|Genuine|
|348702330256514 |37495066290  |4013428|15868   |45845320330319 |2018-06-13 05:38:54|Genuine|
|348702330256514 |37495066290  |5495353|79033   |545499621965697|2018-06-16 21:51:54|Genuine|
|348702330256514 |37495066290  |3966214|22832   |369266342272501|2018-10-21 03:52:51|Genuine|
|348702330256514 |37495066290  |1753644|17923   |9475029292671  |2018-08-23 00:11:30|Genuine|
|348702330256514 |37495066290  |1692115|55708   |27647525195860 |2018-11-23 17:02:39|Genuine|
|5189563368503974|117826301530 |9222134|64002   |525701337355194|2018-03-01 20:22:10|Genuine|
|5189563368503974|117826301530 |4133848|26346   |182031383443115|2018-09-09 01:52:32|Genuine|
|5189563368503974|117826301530 |8938921|76934   |799748246411019|2018-12-09 05:20:53|Genuine|
|5189563368503974|117826301530 |1786366|63431   |131276818071265|2018-08-12 14:29:38|Genuine|
|5189563368503974|117826301530 |9142237|50635   |564240259678903|2018-06-16 19:37:19|Genuine|
|5407073344486464|1147922084344|6885448|59031   |887913906711117|2018-05-05 07:53:53|Genuine|
|5407073344486464|1147922084344|4028209|80118   |1162660511118182|2018-08-11 01:06:50|Genuine|
|5407073344486464|1147922084344|3858369|53820   |896105817613325|2018-07-12 17:37:26|Genuine|
|5407073344486464|1147922084344|9307733|14898   |729374116016479|2018-07-13 04:50:16|Genuine|
|5407073344486464|1147922084344|4011296|44028   |543373367319647|2018-10-17 13:09:34|Genuine|
|5407073344486464|1147922084344|9492531|49453   |211980095659371|2018-04-21 14:12:26|Genuine|
|5407073344486464|1147922084344|7550074|15030   |3455330881120 99|2018-09-29 02:34:52|Genuine|
+----------------+-------------+-------+--------+--------------+-------------------+-------+
only showing top 20 rows
```