Credit Card Fraud Detection System

As per the guidelines provided in the Problem Statement, the solution is broken down into 07 tasks numbered 1 to 7.

Tasks 1 to 4 are designed using the batch-processing approach and Tasks 5 to 7 are designed using the real-time stream processing approach.

Let’s walkthrough the solution.

Task 1: Load the transactions history data (card_transactions.csv) in a NoSQL database and create a look-up table with columns specified earlier in the problem statement in it.

Solution 1:

1. Using HBase Shell, create an HBase table named ***card_transactions*** with a column family named ***cf1*** (createTableForCardTransactions.sql)

2. Using Hive Editor in Hue, create a Hive External table named ***card_transactions*** in the ***default*** Hive database. This Hive table is mapped with the HBase table ***card_transactions***. (createTableForCardTransactions.hql)

3. Using Hive Editor in Hue, create a Hive External table named ***stagging*** in the ***default*** Hive database. This is a stagging table to load data from the ***card_transactions.csv*** file. (createTableForCardTransactions.hql)

   Pre-requisites to step 4: Copy the ***card_transactions.csv*** to a location on HDFS (copyCard_TransactionsCSV.txt)

4. Load the data of ***card_transactions.csv*** in the Hive table named ***stagging***. (loadCardTransactions.hql)

5. Load the entire data from the Hive table **stagging** into the Hive table **card_transactions** (loadCardTransactions.hql)

Task 2: Write a script to ingest the relevant data from AWS RDS to Hadoop.

Solution 2:

Pre-requisites to step 1: Setup some HDFS directories with relevant permissions.(setupDirForDataIngestion.txt)

1. Create an External Hive table named ***card_member*** in the ***default*** Hive database.(createTableCardMember.hql)

2. Create a Sqoop job named ***GetCardMember*** to ingest the data from the AWS RDS table ***card_member*** to the Hive table ***card_member***. This is an incremental sqoop import job. (ingestCardMemberData.sqoop)

3. Create an External Hive table named ***member_score*** in the ***default*** Hive database. (createTableMemberScore.hql)

4. Create a Sqoop job named ***GetMemberScore*** to ingest the data from the AWS RDS table ***member_score*** to the Hive table ***member_score***. (ingestMemberScoreData.sqoop)

   Note: These scoop jobs will be executed as part of oozie workflow.

Task 3: Write a script to calculate the moving average and standard deviation of the last 10 transactions for each card_id for the data present in Hadoop and NoSQL database. If the total number of transactions for a particular card_id is less than 10, then calculate the parameters based on the total number of records available for that card_id. The script should be able to extract and feed the other relevant data ('postcode', 'transaction_dt', 'score', etc.) for the look-up table along with card_id and UCL.

Solution 3:

1. Using HBase Shell, create an HBase table named ***look_up*** with a column family ***cf1*** (createTableForLookUp.sql)

2. Using Hive Editor in Hue, create a Hive External table named ***look_up*** in the ***default*** Hive database. This Hive table is mapped with the HBase table look_up. (createTableLookUp.hql)

3. Using Hive Editor in Hue, create a Hive External table named ***query_stagging*** in the ***default*** Hive database. This is a stagging table which stores the calculated UCL and associated card_id. (createTableQueryStagging.hql)

4. Insert data into the Hive table named ***query_stagging*** using the Hive query which calculates UCL. (loadQueryStagging.hql)

5. Load data into the Hive table ***look_up*** by performing a join on the following tables:
   a) card_transactions
   b) query_stagging
   c) member_score
   (loadLookUp.hql)

Task 4: Set up a job scheduler to schedule the scripts run after every 4 hours. The job should take the data from the NoSQL database and AWS RDS and perform the relevantanalyses as per the rules and should feed the data in the look-up table.

Solution 4:

1. In order to perform the analyis i.e. calculate UCL and update the **look_up** table, an oozie scheduler is written with the following components:
   a) *Workflow*
   b) *Coordinator*
   c) *Job properties*

2. The workflow spawns 02 parallel sqoop actions to fetch data from **card_member** and **member_score** AWS RDS tables respectively:
   a) *sqoopjob_getCardMember*
   b) *sqoopjob_getMemberScore*

3. The above **02** scoop actions are joined into a Hive query action **hivejob_calculateUCL** which calculates the UCL. Once this scoop action is completed, the Hive query action **hivejob_loadLookupTable** is spawned.

4. The Hive query action hivejob_**loadLookupTable** updates the **look_up** table with UCL and other details for a **card_id.**

5. The workflow is scheduled to execute after every 04 hours. The coordinator is configured for this purpose. (oozie/)

For Tasks 5, 6 and 7, a Spark streaming application is designed and developed which will consume real-time streaming data from a Kafka Server. This transaction data will be consumed and processed to validate if a particular transaction is GENUINE or FRAUD based on 03 parameters defined in the problem statement. The categorised transaction data will then be written to the card_transactions table available in a NOSQL database.

Spark Streaming Application Structure

1. *RealtimeFraudDetectionApplication.java*
   - ✔ This is the main class from where the Spark Streaming application is launched.

2. *TransactionPOJO.java*
   - ✔ This is a **POJO** (Plain Old Java Object) class which holds transaction data received from the Kafka server. It contains private member variables which correspond to the transaction details like **card_id, member_id, amount** etc. and a set of public getter/setter methods for accessing/modifying the transaction details.

3. *TransactionDAO.java*
   - ✔ This is a **DAO** (Data Access Object) class which contains methods for connecting to the NOSQL database and other **DML** (Data Manipulation Language) methods for reading and writing data to the database

4. *ZipCodeDistance.java*
   - ✔ This is a utility to calculate the distance in Kilometers between **02** post codes i.e. post code of the current transaction and the last transaction of a card holder

5. *ZipCode.java*
   - ✔ This is a **POJO** which holds zip code data information for a post code. It contains private member variables which correspond to the post code details like latitude, longitude etc. and a set of public getter/setter methods for accessing/modifying the post code details.

Spark Streaming Application Logic Flow

1. **Initialize The Spark Streaming Application**
   - ✔ Create and initialize a Spark Configuration object to run the application in local mode.
   - ✔ Create a HashMap object to store NOSQL database parameters received from command line arguments. This information is passed to the **TransactionDAO** class via the **initializeTransactionDAO()** method.
   - ✔ Create a **JavaStreamingContext** object to create DStreams with an interval of 1 second.
   - ✔ Create a HashMap object to store Kafka parameters to be used for connecting to the Kafka Server.
   - ✔ Using the **createDirectStream()** method of the **KafkaUtil** class, create a JavaInputDStream which will contain transactions data (as JSON strings) as RDDs within Dstreams.

2. **Consume And Process The Transaction Data**
   - ✔ Create a **FlatMapFunction()** to process the transactions data received as JSON strings in RDDs of Dstreams.
   - ✔ Using the **toJSON()** method of the **JSONSerializer** class, parse the JSON strings as JSON objects.
   - ✔ Retrieve and store the transaction data from the JSON objects using the **get()** method.
   - ✔ For each record (transaction), create an object of **TransactionPOJO** class and initialise the transaction data (retrived from the JSON object) using the setter methods of the class.

3. **Evaluate The Transaction Based On 03 Parameters**
   - ✔ **Rule1:** Transaction Amount should be less than or equal to UCL
     - Using the **getUCL()** method of the **TransactionDAO** class, retrieve the UCL from the **look_up** table for the card holder.
     - Using the **getAmount()** method of the **TransactionPOJO** class, retrieve the amount from

the **TransactionPOJO** object containing transaction data.

- Check if transaction amount is less than or equal to UCL. If yes, set ruleUCL boolean variable to true.

✔ **Rule2:** Member Score should greater than or equal to 200

- Using the **getScore()** method of the **TransactionDAO** class, retrieve the member score from the look_up table for the card holder.
- Check if score is greater than or equal to 200. If yes, set ruleScore boolean variable to true.

✔ **Rule3:** Distance travelled between 02 post codes should be greater than 0.25 KM/sec

- Using the **getPostCode()** method of the **TransactionDAO** class, retrieve the zip code of the last transaction from the **look_up** table for the card holder.
- Using the **getPostCode()** method of the TransactionPOJO class, retrieve the zip code of current transaction from the **TransactionPOJO** object containing transaction data.
- Using the **getDistanceViaZipCode()** method of the ZipCodeDistance class, calculate the distance in Kilometers between the 02 zip codes.
- Using the **getTransactionDate()** method of the TransactionDAO class, retrieve the transaction date of the last transaction from the **look_up** table for the card holder.
- Using the **getTransactionDate()** method of the **TransactionPOJO** class, retrieve the transaction date of current transaction from the **TransactionPOJO** object containing transaction data.
- Calculate the difference in time between the last transaction date and current transaction date.
- Calculate the distance covered in secs using the below formula:

distanceCoveredInSecs = distanceInKM/dateDiffenceInSecs

- Check if distanceCoveredInSecs is less than 0.25. If yes, set ruleZipCode boolean variable to true.
- Note: In order to deal with incorrect transaction date data i.e. the last transaction date in the look_up table is greater than or newer than the current transaction date, it was proposed to take an absolute value of transaction dates difference to make sure a lot of data does not get labeled as FRAUD. I have taken the suggested and most popular approach.

4. **Update The NOSQL Database**
   - ✔ If all the rules are met, then classify the transaction to be **GENUINE**
     - Using the **setStatus()** method of the **TransactionPOJO** class, set the status as **GENUINE.**
     - Using the **updateLookUp()** method of the **TransactionDAO** class, update the new post code and transaction date in the **look_up** table.
   - ✔ Else classify it as **FRAUD**
     - Using the **setStatus()** method of the **TransactionPOJO** class, set the status as **FRAUD.**
   - ✔ Using the **insertTransaction()** method of the **TransactionDAO** class, insert the new transaction details in the **card_transactions** table.

Execution Environment Details For Spark Streaming Project

1. This project contains the following JAVA classes:
   - ✔ *RealtimeFraudDetectionApplication.java*
   - ✔ *TransactionPOJO.java*
   - ✔ *TransactionDAO.java*
   - ✔ *ZipCodeDistance.java*
   - ✔ *ZipCode.java*

2. Please make sure JDK1.8 is installed on the machine on which the programs are executed.

3. All of these JAVA classes (and a few more supporting classes) will be bundled in the CreditCardFraudDetection.jar

4. Note: This project is tested ONLY
   - ✔ In Local Mode
   - ✔ On LINUX
   - ✔ Please note this program is Not Tested On Windows OS.

5. Create a folder named data and place the zipCodePosId.csv file in it. This folder should be placed at the same location from where the program is executed.

6. Below is the syntax of the command for running the programs via command-prompt:

   java -cp <Absolute file system path of the JAR file>

7. Below is a sample example, in this case the JAR file is present on the same location where the command prompt is launched:

   java -cp CreditCardFraudDetection.jar