The RDS connection string and credentials are as follows:

* RDS **Connection String** -

jdbc:mysql://upgradawsrds1.cyaielc9bmnf.us-east-1.rds.amazonaws.com/cred\_financials\_data

* **Username** - upgraduser
* **Password** - upgraduser
* **Database -**cred\_financials\_data
* **Table Name** - card\_member and member\_score

To connect to Kafka use the following details:

* **Bootstrap-server:**18.211.252.152
* **Port Number:**9092
* **Topic**: transactions-topic-verified
* Make sure that the Thrift server is running.
* Provide the Public IP of your EC2 Instance in place of the IP address "54.86.61.43" as mentioned in the file shared.

**Tables**

**card\_member**(The cardholder’s data is stored in a central AWS RDS.)

* + card\_id: This refers to the card number.
  + member\_id: This is the 15-digit member ID of the cardholder.
  + member\_joining\_dt: This is the date and time of joining of new member.
  + card\_purchase\_dt: This is the date on which the card was purchased.
  + country: This is the country in which the card was purchased.
  + city: This is the city in which the card was purchased.

**card\_transactions**(All incoming transactions (fraud/genuine) swiped at point of sale (POS) terminals are stored in this table.)

* + card\_id: This refers to the card number.
  + member\_id: This is the 15-digit member ID of the cardholder.
  + amount: This is the amount that is swiped with respect to the card\_id.
  + postcode: This is the ZIP code at which this card was swiped (marking the location of an event).
  + pos\_id: This is the merchant’s POS terminal ID, using which the card was swiped.
  + transaction\_dt: This is the date and time of the transaction.
  + status: This indicates whether the transaction was approved or not, with a genuine/fraud value.

**member\_score**(The member credit score data is stored in a central AWS RDS.)

* + member\_id: This is the 15-digit member ID of the cardholder.
  + score: This is the score assigned to a member defining their credit history, generated by upstream systems.

Data related to **card\_member**and **member\_score**is stored in a central AWS RDS. You will be given the **card\_transactions**data, which has already been classified, in the form of a CSV file, which you can load in your NoSQL database.

The other type of data is the real-time streaming data that is generated by the POS systems in a JSON format. The streaming data looks like this:

* Transactional payload (data) attributes sent by POS terminals’ gateway API on to the Kafka topic:
  + card\_id: This is the card number.
  + member\_id: This is the 15-digit member ID of the cardholder.
  + amount: This is the amount that is swiped with respect to the card\_id.
  + pos\_id: This is the merchant’s POS terminal ID, using which the card was swiped.
  + postcode: This is the ZIP code at which this card was swiped (marking the location of an event).
  + transaction\_dt: This is the date and time of the transaction.

{

"card\_id":**348702330256514**,

"member\_id": **000037495066290**,

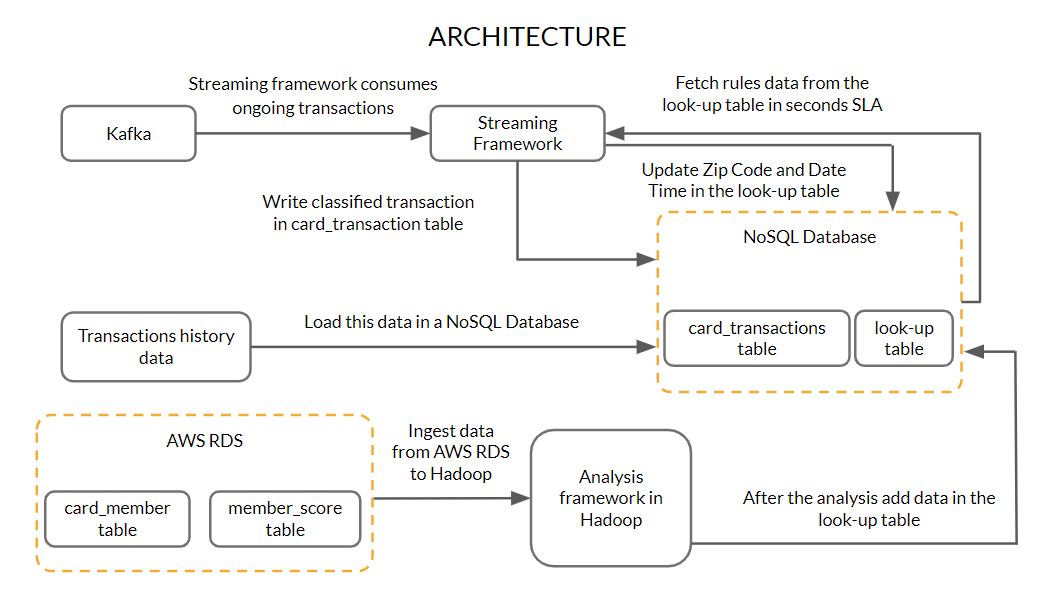
"amount": **9084849**,

"pos\_id": **614677375609919**,

"postcode": **33946**,

"transaction\_dt": "11-02-2018 00:00:00"

}



* The details of the member and the credit score associated with members are hosted on a central AWS RDS server.
* The historical transaction data will be provided as a CSV file.
* You need to use appropriate ingestion methods available to bring the card\_member and member\_score data from the AWS RDS into a Hadoop platform.
* You also need to load the historical card transactions into a NoSQL database.
* This data is then processed to fill data in the look-up table.
* Now, the data from the several POS systems will flow inside the architecture through a queuing system such as Kafka.
* The POS data from Kafka will be consumed by the streaming data processing framework to identify the authenticity of the transactions.
* Once the POS data from Kafka is entered into the stream processing layer, it is then assessed based on some parameters defined by the rules.
* The values for these parameters are fetched from the look-up table.
* The transaction is allowed to complete only when the results are positive for these rules.
* If the result for any rule is negative, then the transaction should be classified as fraud.
* Once the transaction is classified as genuine, then, corresponding to the card ID in the look-up table, the postcode and the transaction date of the current transaction need to be updated as per the last transaction. These fields should only be updated if the transaction gets classified as genuine.
* The card\_transactions table also needs to be updated with all the details along with the classification of the transactions.

The lookup table will contain the following details:

* Card id
* Upper control limit (UCL)
* Postcode of the last transaction
* Transaction date of the last transaction
* The credit score of the member

**Rules:**

**Upper control limit (UCL):**

Every card user has an upper limit on the amount per transaction, which is different from the maximum transaction limit on each card. This parameter is an indicator of the transaction pattern associated with a particular customer.

UCL = Moving average + 3 × (Standard deviation)

This formula is used to derive the UCL value for each card\_id. The moving average and the standard deviation for each card\_id are calculated based on the last 10 amounts credited that were classified as genuine.

**Note:** 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.

**Solution**:

Have a lookup table that stores the UCL values based on the moving average and standard deviation of the last 10 transactions for each card\_id. Whenever a transaction occurs, the record corresponding to the card\_id can be easily fetched from this lookup table

**Amount < ucl**

**2. Credit score of each member:**

These scores are updated by a third-party service. If the score is less than 200, that member’s transaction is rejected, as they could be a defaulter. This rule simply defines the financial reputation of each customer.

**Credit score > 200**

3. **ZIP code distance**: The whole purpose of this rule is to keep a check on the distance between the card owner's current and last transaction location with respect to time. If the distance between the current transaction and the last transaction location with respect to time is greater than a particular threshold, then this raises suspicion on the authenticity of the transaction.

**Solution**:

Whenever a new transaction occurs, retrieve the ‘postcode’ and ‘transaction\_dt’ attributes from the look-up table and compare these with the current ‘postcode’ and ‘transaction\_dt’ data. Use the API to calculate the speed at which the user moved from the origin. If it is more than the imaginable speed, this can be a possible case of fraud.

**Implementation**

You will need to make sure that you have **Hadoop, Sqoop, Hive, HBase and Spark** installed on your cluster with **Hue**as an optional service. Also as an added step, make sure that in the **Hardware configuration step** for the EMR cluster generation, scroll down to the **EBS Root Volume configuration** and type the **Root device EBS volume size** as **20 GB.**

* **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.
* **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.

**Note:**At the end of **three weeks**you are required to make submissions for the **first four tasks**.

**Validation**

1. When you load the data of the past card transactions in the NoSQL database the count of the data should be **53,292**. This will be same as the number of records present in the **card\_transactions.csv** file.
2. When you run the sqoop jobs to import the data from AWS RDS it would retrieve 999 records.
3. When you classify all the incoming transactions as fraud or genuine and then update this in the card\_transactions table, the final count of that table should be more than **59,000**.

Upload a zip file containing the following:

1. ​​​​A PDF document (**LoadNoSQL.pdf**) containing the commands to load the transactions history data (card\_transactions.csv) in a NoSQL database.
2. A PDF document (**SqoopDataIngestion.pdf**) containing the code used for ingesting data from the RDS server.
3. A PDF document (**CreateNoSQL.pdf**) to create a look-up table with columns specified earlier in the problem statement.
4. 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. (**PreAnalysis.pdf**)  
   **Note:** In this pdf provide all the commands to load the lookup table with the relevant data.
5. Screenshots of the execution of the scripts written. The scripts should, after loading the data and creating the look-up table, take the data from the NoSQL database and AWS RDS and perform the relevant analyses as per the rules and should feed the data in the look-up table (**ScriptsExecution.pdf**).
6. Explanation of the solution to the batch layer problem in detail should be provided properly in a document. (**LogicMid.pdf**)  
   **Note: All the sample files have been provided to you as a word document at the end of this segment in a zipped file.**
7. The structure of your directory should be as described ahead. A directory named "**python**" should be created. It should contain a directory named "**src**". The "**src**" directory should have two directories named "**db**" and "**rules**".  The "**src**" directory should have a python file named "**driver.py**" which should be the calling the other files and should be the entry point of your code. The "**rules**" directory should contain a "**rules.py**" file where you write the functions to check for the three rules mention earlier. The "**db**" directory should have the "**geo\_map.py**" and "**dao.py**" shared earlier. The **driver.py**file should contain the code to read the messages from Kafka and call necessary functions from the python files present in the "**rules**"and "**db**"directory to classify the incoming transaction as fraud or genuine.  
   **Note: It is not necessary to follow the above guidelines. Whatever the code you write, you need to properly comment it and also document the exact steps to run the code provided by you.**  
   **Note: It is very important to provide a document to explain the code that you have written for the streaming layer. Also, you need to properly document the steps to execute the same code.**
8. Explanation of the solution to the streaming layer problem in detail should be provided properly in a document. (**LogicFinal.pdf**)  
   **Note: This is a must. You need to clearly explain the code and the steps to run the same should be properly documented.**

-------------------------------------------

Batch: 3

-------------------------------------------

+----------------+-------------+---------+--------+---------------+-------------------+

|card\_id |member\_id |amount |postcode|pos\_id |transaction\_dt |

+----------------+-------------+---------+--------+---------------+-------------------+

|4708912758619517|7955566230397|2413358.0|45864 |878054127728540|2018-07-11 11:43:12|

|4708912758619517|7955566230397|113130.0 |19026 |334571615323048|2018-11-13 03:36:30|

|4708912758619517|7955566230397|6680657.0|30417 |666504956332883|2018-08-28 11:46:08|

|4708912758619517|7955566230397|3270338.0|29809 |630643318258525|2018-10-28 06:20:00|

|4708912758619517|7955566230397|9331876.0|71369 |664536700035529|2018-05-30 00:44:36|

|4708912758619517|7955566230397|6324265.0|40830 |29683308794434 |2018-07-30 07:16:10|

|5342400571435088|8732267588672|190337.0 |84037 |243685870332181|2018-09-10 06:17:03|

|5342400571435088|8732267588672|5354532.0|61924 |285038508626445|2018-02-12 04:33:29|

|5342400571435088|8732267588672|3190064.0|99774 |440589004127017|2018-10-29 18:57:56|

|5342400571435088|8732267588672|2989713.0|43318 |833376948788696|2018-03-30 21:52:43|

|4237648081700588|8765307152821|9481368.0|55367 |443273719811115|2018-10-01 10:39:04|

|4237648081700588|8765307152821|7149819.0|16629 |139814927796717|2018-12-02 05:08:19|

|4237648081700588|8765307152821|6070972.0|25253 |710246811254202|2018-12-05 19:51:36|

|4237648081700588|8765307152821|899958.0 |57245 |676738830318587|2018-11-20 21:00:46|

|4237648081700588|8765307152821|4112190.0|15447 |370316491310138|2018-08-27 11:05:53|

|371814781663843 |9136568025042|5288853.0|12810 |177709663894455|2018-07-02 22:52:41|

|371814781663843 |9136568025042|6358817.0|75491 |446324941465834|2018-10-16 22:12:27|

|371814781663843 |9136568025042|699943.0 |72638 |233669422387972|2018-09-17 18:27:19|

|371814781663843 |9136568025042|8058055.0|11932 |224392627656727|2018-06-23 06:22:30|

|371814781663843 |9136568025042|9942239.0|30039 |343255811642668|2018-07-30 21:28:25|

+----------------+-------------+---------+--------+---------------+-------------------+