Problem Statement

Imagine you are working for a leading credit card company called ‘Cred Financials’. The company continuously monitors its customers’ credit card transactions, be it in any part of the world, to discover and dismiss fraudulent ones. The company also has a strong support team to address customer issues and queries.

Credit card fraud is defined as a form of identity theft in which an individual uses someone else’s credit card information to make purchases or to withdraw funds from the account. The incidences of such fraudulent transactions have skyrocketed as the world has moved towards a digital era.

With a rising number of fraud cases, the company’s major focus is to provide its customers with a delightful experience while ensuring that security is not compromised.

You, as a big data engineer, must architect and build a solution to cater to the following requirements:

Fraud detection solution: This is a feature to detect fraudulent transactions, wherein once a cardmember swipes his/her card for payment, the transaction should be classified as fraudulent or authentic based on a set of predefined rules. If fraud is detected, then the transaction must be declined. Please note that incorrectly classifying a transaction as fraudulent will incur huge losses to the company and also provoke negative consumer sentiment.

Customers' information: The relevant information about the customers needs to be continuously updated on a platform from where the customer support team can retrieve relevant information in real time to resolve customer complaints and queries.

References
https://legaldictionary.net/credit-card-fraud/

Data

Now, let’s understand the types of data you will deal with.

The following tables containing data come into consideration for this problem:

card_member (The cardholder data is added to/updated in this table by a third-party service)

card_id – Card number,

member_id – 15-digit member ID of the cardholder,

member_joining_dt – Date of joining of a new member,

card_purchase_dt – Date and time when the card was purchased,

country – Country in which the card was purchased,

city – City in which the card was purchased

card_transactions (All incoming transactions(fraud/genuine) swiped at POS terminals are stored in this table. Earlier the transactions were classified as fraud or genuine in a traditional way. However, with an explosive surge in the number of transactions, a Big Data solution is needed to authenticate the incoming transactions and enter the transaction data accordingly):

card_id – Card number,

member_id – 15-digit member ID of the cardholder,

amount – Amount swiped with respect to the card_id,

postcode – Zip code at which this card was swiped (marking the location of an event),

pos_id – Merchant’s POS terminal ID, using which the card has been swiped,

transaction_dt – date and time of the transaction event,

status – Whether transaction was approved or not, with Genuine/Fraud value

member_score (The member credit score data is added to / updated in this table by a third-party service):

member_id – 15-digit member ID who has this card,
score – The score assigned to a member defining his/her credit history, generated by upstream systems

Since card_member and member_score tables are updated by the third-party services, they are stored in a central AWS RDS. You will be given the already classified card_transactions table data 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 generated by the POS (Point of Sale) systems in 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 – Card number,

member_id – 15-digit member ID of the cardholder,

amount – Amount swiped with respect to the card_id,

pos_id – Merchant's POS terminal ID, using which the card has been swiped,

postcode – Zip code at which this card was swiped (marking the location of an event),

transaction_dt – date and time of the transaction event

Here is an example of a JSON payload structure that gets produced:

{

"card_id":348702330256514,

"member_id": 000037495066290,

"amount": 9084849,

"pos_id": 614677375609919,

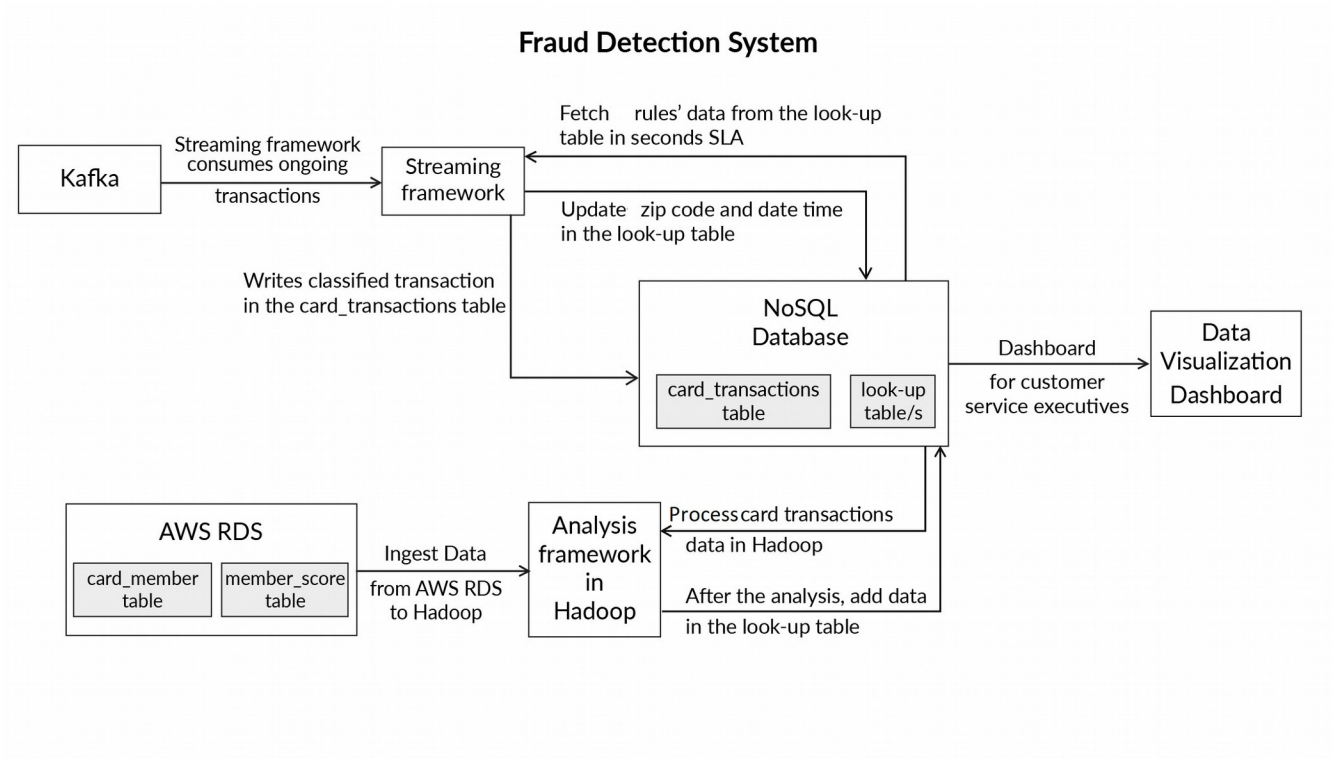
"postcode": 33946,

"transaction_dt": "11-02-2018 00:00:00"

}

Architecture and Approach

Having understood the various kinds of data involved in this project, it's time to understand how to approach the task of building a solution to this problem statement. The following diagram will help you understand how the entire architecture should look like. Read ahead to understand more.



The data from the several POS systems will flow inside the architecture through a queuing system like Kafka. The POS data from Kafka will be consumed by the streaming data processing framework to identify the authenticity of the transactions.

Note: One of the SLAs of the company is to complete the transaction within few seconds. Hence, the framework should be accordingly chosen to facilitate this SLA.

Once the POS data enters into the Stream processing layer, it is assessed based on some parameters defined by the rules. Only, when the results are positive for these rules, the transaction is allowed to complete. Now, what are these rules? How can one obtain these parameters and where are they stored? These questions will get answered in some time.

Once the status of a transaction is determined as a "Genuine" or "Fraudulent", the details of the transaction, along with the status, are stored in the card_transactions table.

Now, let's understand the various parameters defined by the rules required to determine the authenticity of the transactions. Here are the three parameters that we will use to detect whether a transaction is fraudulent or not.

1. Upper Control Limit: Every card user has an upper limit on the amount per transaction that is different from the maximum transaction limit on each card. This parameter is basically an indicator of transaction pattern associated with a particular customer. This upper bound, also known as the “Upper Control Limit” or UCL, can be used as a parameter to authenticate a transaction. Suppose you have a past record of making transactions with an average amount of $20,000, and one day the system observes a transaction of $200,000 through your card. This can be a possible case of fraud. In such cases, the cardholder receives a call from the credit card company executives to validate the transaction. UCL is derived using the following formula:

UCL = (Moving Average) + 3 × (Standard Deviation)

The above formula is used to derive the UCL value for each card_id. The Moving average and the Standard Deviation for each card_id is calculated based on the last 10 amount credited with a ‘Genuine’.

2. Credit score of each member: This is a straightforward rule, where we have a member_score table in which member ids and their respective scores are available. These scores are updated from a third-party service. If the score is less than 200, that member’s transaction is rejected as he/she could be a defaulter. This rule simply defines the financial reputation of each customer.

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. Suppose at time t = t0 minutes, a transaction is recorded in Mumbai and at time t = (t0 + 10) minutes, a transaction from the same card_id is recorded in New York. A flight flies with a cruising speed of about 900km/hr which means that someone travelling by Airbus can travel a KM in 4 Secs, KM=4 Sec. This can be a possible case of fraud. Such cases happen very often when someone acquires your credit card details and make transactions online using those details. In such cases, the cardholder receives a call from the credit card company executive to validate the transaction.

Use the postcode library (will be provided ahead for checking distance between two zip codes) to get the distance between two zip codes.

Now that you know each of the parameters, let’s understand the approach to calculate these.

Let’s start with the upper control limit (UCL). The historical transactional data is stored in the card_transactions table, as defined earlier in the table description. UCL value has to be calculated for each card_id for the last 10 transactions. One approach could be to trigger the computation of this parameter for a card_id every time a transaction occurs. However, considering the few seconds SLA, this may not be a very good practice as batch jobs are always associated with huge time delays.

Another approach could be to have a lookup table which stores the UCL value based on the moving average and standard deviation of the last 10 transactions of each card_id. Whenever a transaction occurs, the record corresponding to the card_id can be easily fetched from this lookup table rather than calculating the UCL value at the time of the transaction. This lookup table needs to be updated at regular intervals by running queries on data stored in the AWS RDS and the NoSQL database.

Note: You need to use a NoSQL distributed database to implement the look-up table. The database must be scalable and consistent. Use a NoSQL database which gives schema evolution, schema versioning, row-level lookups (efficient reads), and tunable consistency. For every ‘card_id’, this database must store the UCL value.

Use appropriate ingestion methods available to bring card_member and member_score data from AWS RDS and card_transactions data from the NoSQL database into the Hadoop platform. This data is then processed by running batch jobs to fill data in the look-up table. After the initial load, there will be incremental loads that should be considered before designing the architecture of the solution.

Once the transactions data is imported into Hadoop, batch jobs are run on the data stored to calculate the moving average and standard deviation of the last 10 transactions for each card_id. Once the moving average and standard deviation are obtained, the UCL value for each card_id is calculated. Then the relevant data is entered in the look-up table.

Note: While wrangling data, keep the stages in place for better backtracking of your transformations (keeping track of steps) as a best practice. Let’s suppose that you are calculating UCL in Hive. You might first need a raw table, then in the next staging table, you might want to derive the moving average and standard deviation and then the final table with UCL value. This helps manage the layers of your wrangled data. This is not mandatory but recommended.

This UCL value is then updated in the look-up table. Since the transactions_table will be continuously incremented with the new transactions, it’s important that UCL value gets continuously updated at regular intervals. For this problem statement, the UCL value will be updated after every 4 hours in the look-up table. This implies the following:

The added data needs to be processed every 4 hours from the transactions_table in NoSQL database to Hadoop

A batch job needs to be run to calculate the UCL value every time the data has to be processed
The new UCL value needs to be updated into the lookup table every time the new UCL value is calculated for each card_id

The second parameter, i.e., the credit card score also gets updated daily by a service in the RDBMS for each card_id. You need to update the credit card score in the same look-up table where the UCL value corresponding to each card_id is getting stored. This job needs to be triggered after every 4 hours (the interval after which the credit card score might also get updated in the RDBMS by the service).

The third parameter is based on the zip code analysis. Store the ‘postcode’ and ‘transaction_dt’ parameters pertaining to the last transaction of each card_id in the look-up table. 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 ahead of the imaginable speed, this can be a possible case of fraud. In such cases, the cardholder receives a call from the credit card company executive to validate the transaction. You also need to update the last transaction data in the lookup table with the current data after the comparison is complete. This has to be done in two steps:

Initially, at one-time load, you need to fetch the latest postcode and transaction_dt of each member and store in NoSQL database with other parameters discussed above

After initiating the real-time process following each member’s transaction, update the current received transaction’s postcode and transaction_dt as the last zip code and time in the lookup table stored in the NoSQL database if and only if the transaction is approved (satisfying all 3 rules)

Once a transaction is evaluated based on the above three parameters, the transaction, along with the status (i.e Genuine or Fraud) of the transaction, is stored in the card_transactions table in the database.

Apart from ‘Genuine’ and ‘Fraud’, a transaction is to be additionally labelled as ‘Suspect’ if it has either of the following characteristics:

1. If a customer has visited a merchant (pos_id) for the first time, mark the transaction as 'SUSPECT'.

2. Consider the frequency of usage of the card based on past 100 transactions. If the time lapse between the current transaction and the last transaction is more than 5 times the average time lapse between transactions, mark the current transaction as 'SUSPECT'.

Now let’s approach the second problem, i.e., building the infrastructure to answer user queries. The infrastructure has to be built to provide details pertaining to the last 10 transactions for each card_id. Querying the card_transactions table every time to address customer’s queries is not efficient, as the data size is large. Having the last 10 transactions’ data in the look-up table for each card_id can result in better customer support service.

Thus, each record in the look-up table must store the following attributes:
‘card_id’,
‘UCL’,
‘postcode’,
‘transaction_dt’,
‘score’.

Note: You need not store ‘last_ten_transaction_details’ in the look-up table as they can be directly fetched from the master table in the NoSQL database.

The other details such as member_id, member_joining_dt, card_purchase_dt, country and city should be also stored for getting customer's information (dashboard for customer care executives).

Once you start the Kafka consumer in the streaming framework, each transaction of different members will be iterated and checked for these rules without any lag.

Guidelines

As part of the project, broadly you are required to perform the following tasks:

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.

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

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. Write a script to calculate the average time lapse between transactions for each card based on its 100 latest transactions. If the total number of transactions for a particular card_id is less than 100, then calculate the parameter based on the total number of records available for that card_id.

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 relevant analyses as per the rules and should feed the data in the look-up table.

Task 5: Create a streaming data processing framework which 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 above.

Task 6: Update the transactions data along with the status (Fraud/Genuine along with Suspect if applicable) 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.