Fraud Transaction Detector in real time using Kafka

N.Ravi Teja 2020H1030143H, Saptarsi Saha 2020H1030109

I. Introduction

A. Motivation

Fraud attempts have seen a drastic increase in recent years, making fraud detection more important than ever. Despite efforts on the part of the affected institutions, hundreds of millions of dollars are lost to fraud every year. Since relatively few cases show fraud in a large population, finding these can be tricky.In banking, fraud can involve using stolen credit cards, forging checks, misleading accounting practices, etc.Detecting fraudulent transactions is one of the classic use cases for real-time data.Modern streaming data technologies like Apache Kafka can help companies catch and detect fraud in real time .Kafka is ideal for managing fast, incoming data points.

B. Work Contributions

Research about Fraud transactions, transaction generator-RaviTeja
Setting up kafka cluster, fraud transactions detection-Saptarsi Saha

II.Background work

A.Event streaming

Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events.Event streaming ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.Kafka combines three key capabilities for event streaming:

1. To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.
2. To store streams of events durably and reliably for as long as you want.
3. To process streams of events as they occur or retrospectively.

B.Basic Concepts

Events: An event is anything that takes place in the world. Software applications generate events every time. A user browsing a social media website sends all its activity information to the backend system. Systems can gather statistics from the events data & extract patterns. For eg:- Netflix can recommend new movies to the users based on previously watched ones.

*Topic:*A topic is a logical aggregation of events. A group of events having some common characteristic becomes a topic. A News website may categorise all the events happening into topics like Politics, Sports, Entertainment, Technology, etc.

*Broker:*The broker is an application which persists the events or messages. Publishers can use APIs to publish messages to a broker. The broker stores messages in sequential order. It's similar to a write-ahead log (WAL). A log file is stored on the disk. As soon as a new message is published, the broker appends a new entry to the end of the file.The broker is an application which persists the events or messages. Publishers can use APIs to publish messages to a broker. The broker stores messages in sequential order. It's similar to a write-ahead log (WAL). A log file is stored on the disk. As soon as a new message is published, the broker appends a new entry to the end of the file.

Publisher: A Publisher is a system that communicates events to a broker. Once the message is published, the Publisher doesn't care about what happens to it next. In this manner, Publisher and the consumer are decoupled from each other.A publisher can be an upstream server application that is producing logs or a client application sending user website activity.

Partition: A single broker system is not scalable as it's a single point of failure. Further, high throughput load can't be served by a single broker. For scaling the messaging system, Kafka uses the concept of partition.The partitioning concept is similar to database partitioning. The core idea is to distribute the data on multiple brokers instead of using a single broker. A topic is divided into partitions. Each partition is managed by a different broker. In case of a single broker failure, messages from one partition will become unavailable. However, messages from other partitions can still be read.Each event can be thought of as a Key-Value pair. Producers can use multiple strategies for distributing the Keys among partitions. A common and efficient approach is Consistent Hashing. Although, others such as Hashing and range-based partitioning can still be used.

*Replication:*Replication is the process of having multiple copies of the data for the sole purpose of availability in case one of the brokers goes down and is unavailable to serve the requests.In Kafka, replication happens at the partition granularity i.e. copies of the partition are maintained at multiple broker instances using the partition's write-ahead log.A replication factor of 2 means that there will be two copies for every partition.For every partition, there is a replica that is designated as the leader**.** The Leader is responsible for sending as well as receiving data for that partition. All the other replicas are called the in-sync replicas (or followers) of the partition.In-sync replicas are the subset of all the replicas for a partition having the same messages as the leader.

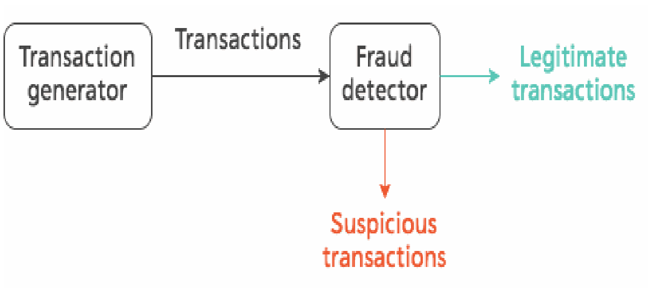
Consumer groups:A consumer group can subscribe to one or more topics. Each consumer group has at least one consumer. Every consumer reads data from a topic partition using an offset. The messages in the partition are not deleted as soon as the consumer reads the data. Hence, those are still available for other consumer groups to consume. Every time a client reads data at a given offset, it’s offset is incremented. Further, the consumer will read data at the next offset.

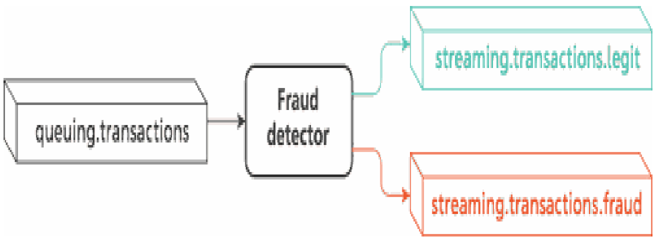
*Zookeeper:*Zookeeper maintains a list of all the brokers running in the cluster. This list is updated whenever a new broker joins or leaves the system. It tracks the topics, number of partitions assigned to those topics, & replica location. It also manages the access control list to different topics in the cluster.

Message queue:A Message Queue works like a queue data structure. Producers add messages to the rear of the queue. Consumers read the messages from the front of the queue.

III.System Architecture and Design

A transactions generator, on one end, which produces fictitious transactions to simulate a flow of events.A fraud detector, on the other end, to filter out transactions which look suspicious transactions.





Topics:

The streaming.transactions.legit contains the legit transactions, streaming.transactions.legit contains the fraudulent transactions and queuing.transactions contains the raw generated transactions.

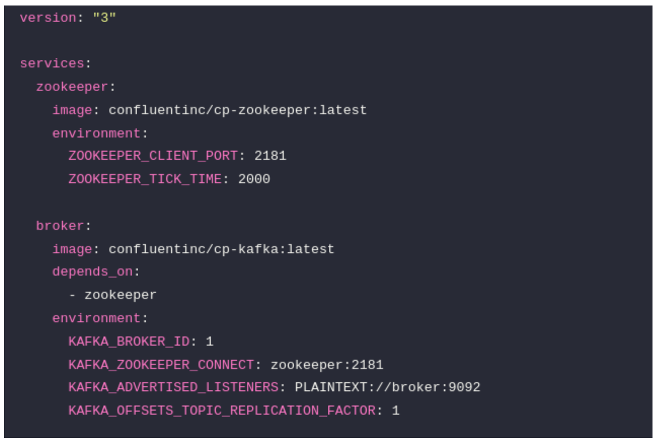
IV.Implementation

A.Docker compose configuration

It has two services: one for the Kafka broker and one for the Zookeeper instance.

Zookeeper: It uses Confluent Platform's Docker image, cp-zookeeper. The two environment variables define which port Zookeeper will be accessible on (which we'll pass to the broker), and its basic unit of time (a required configuration detail).

kafka broker: KAFKA_ZOOKEEPER_CONNECT, which tells the broker where it can find Zookeeper, and KAFKA_ADVERTISED_LISTENERS, which defines where we'll be able to connect to the broker from other applications .



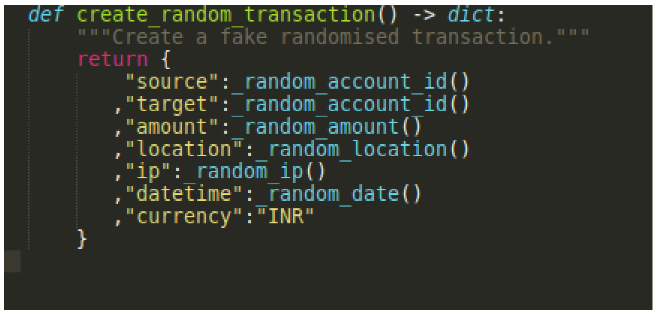
```
version: "3"

services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://broker:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

B.*Building the transaction generator*

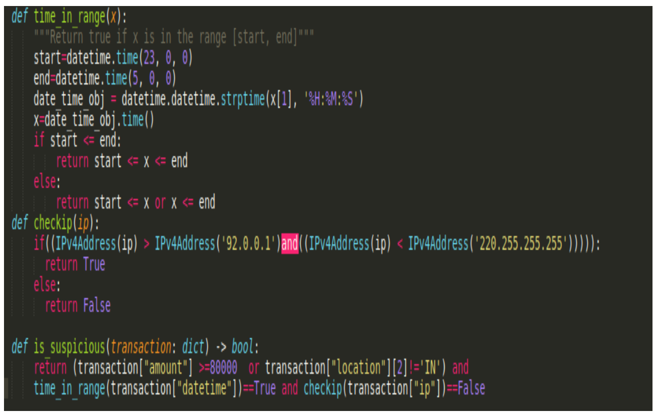
The transaction generator generates random transactions in which each transaction contains from_account number, to_account number, amount, latitude and longitude, country code, date and time of transaction, ip address and currency.



```
def create_random_transaction() -> dict:
    """Create a fake randomised transaction."""
    return {
          "source":_random_account_id()
         ,"target":_random_account_id()
         ,"amount":_random_amount()
         ,"location":_random_location()
         ,"ip":_random_ip()
         ,"datetime":_random_date()
         ,"currency":"INR"
    }
```

C.Defining a Fraud transaction

We define a transaction as fraud if
1)Amount>=80000
2)Country code other than India(IN)
3)Time of transaction is between 23:00:00 to 04:00:00
4)Ip address is not in the range 92.0.0.01 to 220.255.255.255



```
def time_in_range(x):
    """Return true if x is in the range [start, end]"""
    start=datetime.time(23, 0, 0)
    end=datetime.time(5, 0, 0)
    date_time_obj = datetime.datetime.strptime(x[1], '%H:%M:%S')
    x=date_time_obj.time()
    if start <= end:
        return start <= x <= end
    else:
        return start <= x or x <= end
def checkip(ip):
    if((IPv4Address(ip) > IPv4Address('92.0.0.1')and((IPv4Address(ip) < IPv4Address('220.255.255.255'))))):
      return True
    else:
      return False

def is_suspicious(transaction: dict) -> bool:
    return (transaction["amount"] >=80000  or transaction["location"][2]!='IN') and
    time_in_range(transaction["datetime"])==True and checkip(transaction["ip"])==False
```

V.Limitations and Conclusions

Although lots of research has been investigated in the credit card fraud detection field, there are none or limited adaptive techniques which can learn the data stream of transactions as they are conducted. Such a system can update its internal model and mechanisms over a time without need to be relearned offline. Therefore, it can add novel frauds (or normal behaviors) immediately to model or learn fraud tricks and detect them afterward as soon as possible.

A local Kafka cluster that acts as a centralised streaming platform.A transaction generator which produces transactions to the cluster.A fraud detector which processes transactions, detects potentially fraudulent ones and produces the results in two separate topics.There are many ways in which this system could be improved or extended. Here are just a few ideas:
1)Implement a better fraud detection algorithm
2)Send alerts or reports about fraudulent transactions somewhere: email, notifications.