Fraud detection using Neo4j DBMS

Introduction

Banks and Insurance companies lose billions of dollars every year to fraud. Traditional methods of fraud detection play an important role in minimizing these losses. However, increasingly sophisticated fraudsters have developed a variety of ways to elude discovery, both by working together and by leveraging various other means of constructing false identities. In this project, efforts have been made on a simple scenario to detect the fraud transaction for every terminal. Explanation of each part of the problem and the designed solution can be found.

Scenario

The project proposal assumed a small(and simple) part of the banking system to detect the fraud transactions. According to the project proposal there are three entities in the scenario:

- **Customer**: Has a unique id, location and some more properties about his spending style which is the holder of money and spends it on different places.
- **Terminal**: Has a unique id and location which is the place that customers pay their money for different purposes.
- **Transaction**: Has unique id, customer id, terminal id and amount of the transaction which determines source and destination of money.

According to the top assumptions, the UML class diagram of the domain is below image(diagram 1).

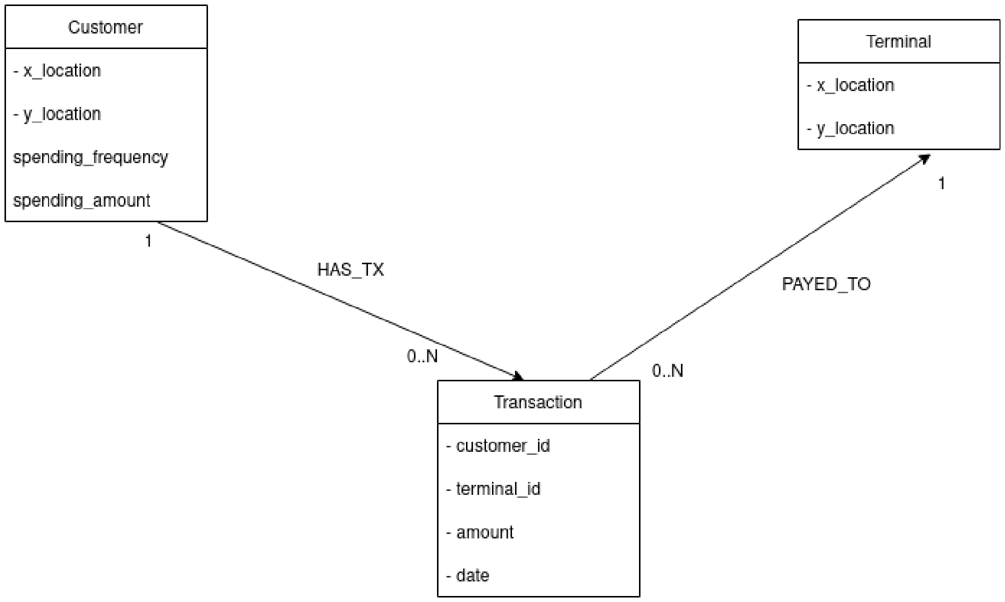

Diagram.1

In the above diagram each entity has its own unique id(is not shown) and there are association relationships between entities. This is not a very similar sample of real world entities but provides enough information to work with and detect fraud transactions. Of course, having information is not enough because there is huge amount of information related to transactions and working with them is very time consuming and in some cases in might not be possible to handle them in the classical relation database management systems(RDBMS) like MySql, PostgreSql and SqlServer, so there is need for new generation databases(NoSql) to deal with these kind of information and operations.

Introduction to Neo4j

Neo4j is a nosql graph database management system developed by Neo4j, Inc. Described by its developers as an ACID-compliant transactional database with native graph storage and processing, Neo4j is available in a GPL3-licensed open-source "community edition", with online backup and high availability extensions licensed under a closed-source commercial license. Neo also licenses Neo4j with these extensions under closed-source commercial terms.
Neo4j is implemented in Java and accessible from software written in other languages using the Cypher query language through a transactional HTTP endpoint, or through the binary "Bolt" protocol.

Chosen NoSQL database

To achieve the project goals, the chosen database is **neo4j** with respect to its use cases, the best practices of this domain and its most important power to handle relationships between entities(that is exactly the project goal) and unstructured data. In diagram 2 you can see the designed logical data model for this scenario which is inspired by Max De Marzi blog(On the blog scenario was about the flight system).

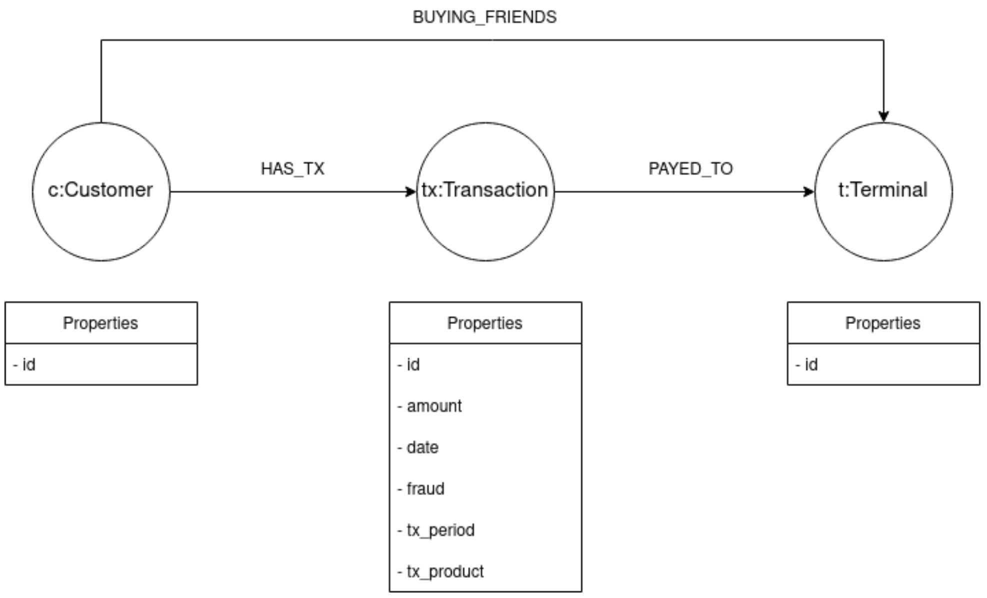

Diagram.2

According to diagram 2 there are 3 nodes and 3 relationships between these nodes, so it makes it much easier to find all related or correlated customers to a terminal. Moreover, properties of the transactions allow us to query on different periods, but there is just an id for the other nodes because their information is useless to satisfy the operations, so to keep the database as much as possible small and fast they have been omitted.

Scripts Examination

In this section we will dive into provided scripts and explain each of them until we reach the result of the requested operations. There are 4 main scripts that have different responsibilities, "**generator.py**" generates datasets with different sizes and saves them as **CSV** format. "**loader.py**" load generated datasets by previous script to the neo4j database, "**updater.py**" update records with "**periods**" and "**products**" then add buying friends relationship and the last one "**main.py**" send the query requests to neo4j for the correspondent database and shows the result.

Prerequisites

There is some work to make the environment ready and run the scripts. First of all, you need to download the **neo4j-desktop** (because the online free version has limits on nodes and relationships) application from this link and install it on your machine. Secondly, download and install **python 3.8+** and **pip3** on your environment. Then, you need to install the required packages that are specified in the **"requirements.txt"** file and create 3 databases on your neo4j database using neo4j-desktop, now your environment is ready to run the scripts, so in next sections explanations to run each script is provided. Timing of each part is measured by the **timeit decorator** that is implemented in "**timing.py**", total timings are mentioned but for more specific details of timing of each section you can refer to the "**log.txt**" file.

Generate Datasets

To generate datasets the same method as suggested in the proposal used from the fraud detection hand book so most of the functions are the same except the **"add frauds function"** and format of output files. In the project scenario we look for the transaction which is more than 50% of last month's average, to make this happen the ability to add these kinds of fraud transactions has been added to the function. Hence, Datasets are generated and provided already(3 datasets generated with 68, 157 and 230 MB size), you can skip this part to the next section. However, to generate a new dataset with a different size in the function "**main()**" there are some factors that determine the number of customers, terminals and days to generate the transactions.
Timing for this part is like this:

- First dataset(68 MB): Total time to generate dataset is ~ 93 seconds.
- Second dataset(157 MB): Total time to generate dataset is ~ 331 seconds.
- Third dataset(230): Total time to generate dataset is ~ 363 seconds.

Loading data into database

Script “**loader.py**” is responsible for loading the data into the database and creating nodes and relationships in neo4j. Inside the “**Settings.py**” you can find different settings for each part of the program, most of them doesn’t need any change but about the database related settings, here you can specify your database url, your credential and database name. Then you can run this script and wait for it to fill the neo4j database with the datasets. To increase the performance and speed of loading datasets instead of running each query separately, the script runs them in a transaction together, in other word, first split them in small **chunks** and run them in a transaction to database then goes to the next chunk, running create queries in this way increase performance more than **10 times**. Before applying this way, filling the database with 5000 nodes took ~ **1 minute** but after, it took only **8 seconds**. Beside this approach, 2 indexes are created on **id** property of 2 Labels(or node types: Customer and Terminal) to speed up creating relationships between nodes(because to create a relationship first finds it, so with indexes it searches on a **binary** tree).
Here you can see timing for this part:

- First dataset: Total time to generate dataset is ~ 1,105 seconds.
- Second dataset: Total time to generate dataset is ~ 2,161 seconds.
- Third dataset: Total time to generate dataset is ~ 3,230 seconds.

Updating database

Adding new values to each transaction(periods and products) is the job of the “**updater.py**” script. This script search does updating in some small steps, first with the **limit** keyword in cypher fetch some rows that don't have the product column, then update them with a random value. Finally, by using **chunks** approaches try to add **buying friends** relationships between customers and terminals(**merge** is used to prevent creating multiple similar relationships). Hopefully, updating is much faster than creating operations and here you can see the timing:

- First dataset: Total time to generate dataset is ~ 16 seconds.
- Second dataset: Total time to generate dataset is ~ 143 seconds.
- Third dataset: Total time to generate dataset is ~ 265 seconds.

Querying database

The “**main.py**” script is responsible for running queries and showing their result. To see the result of this script you can set the database name in “**settings.py**” and run the script. There are 4 main queries that is discussed below separately:

- **Daily customer payments in this month**: The dataset is not real-time, so the dates are now the same as current date so to get the result of this query, a period containing start and end date is given to the function then it groups and sorts(asc) the dates and returns to the user. However, for real-time data we could use **date()** and **duration()** built-in cypher functions to calculate the result. This is not a complicated query so it doesn’t take too much to run, here is the timing for it:
    - First dataset: Total time to generate dataset is ~ 300 milliseconds.
    - Second dataset: Total time to generate dataset is ~ 2.7 seconds.

    - Third dataset: Total time to generate dataset is ~ 8.7 seconds.
- **Terminal fraud transactions**: To get the total number of queries for each terminal we will face a polynomial space and time complexity query **O(30n^2+3t)** which **n** is the number of days and **t** is total number of transactions(query provided in "**queries.tx**" file), so for the our dataset this is a huge query and runs forever(after an hour will be terminated by the engine because memory heap size). However, to solve this problem a period of date(1 month) is assumed and passed as argument to the function to calculate the fraud transactions in a specific period of dates.
    - First dataset: Total time to generate dataset is ~ 500 milliseconds.
    - Second dataset: Total time to generate dataset is ~ 11 seconds.
    - Third dataset: Total time to generate dataset is ~ 21 seconds.
- **Co-customer relationship with degree k**: This is the easiest query in neo4j DBMS because it is made exactly to handle these kinds of queries. As you can see below, the result of this query is not related to the size of the database if used with limited results.
    - First dataset: Total time to generate dataset is ~ 0.04 seconds.
    - Second dataset: Total time to generate dataset is ~ 0.03 seconds.
    - Third dataset: Total time to generate dataset is ~ 0.07 seconds.

  As demonstrated In the figure.1 you can see result of CC relationship for a customer with degree 4(four terminals) which is limited to 100 customers, in the figure the **red** circles are **customers**, **greens** are **transactions** and the **light** circles are **terminals**.

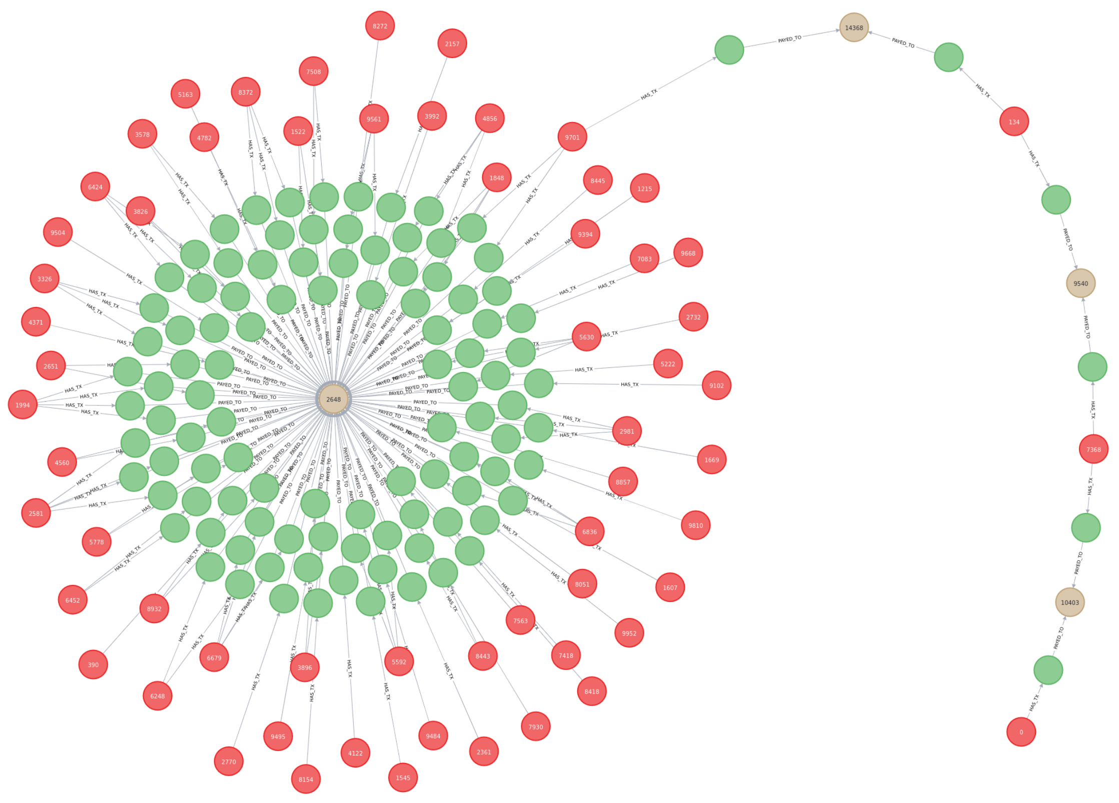

Figure.1

- **Transactions count of each period of day**: This query is the extended version of the second query with grouped results by **period** property.

- First dataset: Total time to generate dataset is ~ 11 seconds.
- Second dataset: Total time to generate dataset is ~ 12 seconds.
- Third dataset: Total time to generate dataset is ~ 22 seconds.

Size of the database

By filling the database with provided CSVs size of the database according to the relationships and nodes grows rapidly because it holds many more informations about them and their indexes so finally size of the database with 68 MB dataset will be **1.8 GB**, database with 157 MB will be **4 GB** and database with 230 MB will be **7 GB**.

Alternative patterns

There are many different scenarios and approaches to detect the fraud transactions according to the context, in this section some patterns will be discussed to achieve the most efficient result of queries. Of course, running these kinds of big queries on an under pressure database is not efficient, these queries can be limited and splitted into some small queries(as done is previous steps) then run on low stress periods of day(e.g. midnight) and save the result another place to be consumed later. Another pattern that can lead to real-time fraud detection is to check each transaction exactly after it is created in an asynchronous way and mark it with the result. In addition, to improve the performance of the queries for sure we don't need transactions for more than 1 year ago, so those transactions can be archived to prevent them participating in new queries.

Conclusion

While no fraud prevention measures can ever be perfect, a significant opportunity for improvement lies in looking beyond the individual data points, to the connections that link them.
Using neo4j to handle these connections between different individuals can be very useful and help to detect the fraud transaction much faster and easier in comparison with other approaches or DBMS. To sum up, fraud detection can be considered as one of the main use cases for the neo4j DBMS.