→ Explanation of the PG data model chosen - details, reason why

Nowadays data are in motion, change continuously and are -possibly- unbounded implying data sources that are also constantly evolving. From the data persistence point of view this reality breaks the usual paradigm of having dynamic but stable data sources. This, together with the increasing number applications based on data streams for taking critical decisions in real time, raises the need for re-thinking both the data and the query models to fit these new requirements. Therefore, under these circumstances, it seems reasonable that a suitable data model is a continuously evolving data graph.
... Regarding the data model, the new nature of data requires a de facto new database paradigm -continuously evolving databases- where data can be both stable and volatile. Even though evolving databases can be implemented according to any approach, graph databases seem especially well suited here [1, 2]. Indeed, the natural way to process evolving graphs as streams of edges gives insights on how to proceed in order to maintain dynamic graph databases. Hence, we consider that a suitable data model is a continuously evolving data graph, a graph having persistent (stable) as well as non persistent (volatile) relations. Stable relations correspond to edges occurring in standard graph databases while volatile relations are edges arriving indata streams during a set time interval. Once this time interval is over, the relations are not longer valid so that there is no need to store them in the (stable) graph database. However, when required -as for further legal or auditing purposes- timestamped occurrences of volatile relations can be kept in a log file. Volatile relations induce subgraphs that exist only while the relations are still valid. Without loss of generality, in this work we consider property graphs (PG) [3, 4] as the basic reference data model. As an example, Figure 1a depicts part of a schema of a PG database where stable relations correspond to the data that a bank typically gathers on its issued cards, ATMs (Automated Teller Machines) network, etc. Volatile relations model the interaction between cards and ATM entities

The property graph data model consists of two sub property graphs: a stable and a volatile property graph. On the one hand, the stable is composed of the most static part of the data that a bank typically gathers such as information about its clients, cards, ATMs (Automated Teller Machines). On the other hand, the volatile property graph models the transaction operations, which defines the most frequent and reiterative kind of interaction between entities of the data model.
The main difference and the main reason for this separation is the semantics with which we intentionally define each of the subgraphs: the stable will be understood like a fixed static bank database, whereas the volatile will be understood as the data model to define the transactions, as continuous interactions between the entities of the model, which won't be permanently saved, but instead, only for a certain window of time under the mission of detecting anomalous bank operations. Note that we will only model the transaction interaction in the volatile subgraph, only letting them occur here. This separation will allow us to have a really simple and light property graph schema single-centered on the transactions with the minimal needed information (mostly identifiers of the en-

ties a transaction links) and another, the stable, acting as a traditional bank database schema, from which to obtain the information details of the entities.

0.1 Property Graph Design

In what follows we provide the description of the design of our proposed Property Graph data model, divided into the stable and volatile property graphs.

0.1.1 Stable Property Graph

As mentioned, due to the confidential and private nature of bank data, it was impossible to find a real bank dataset nor a real bank data model. Therefore, we did our own proposal of a bank database model. We propose a simplified data model where the defined entities, relations and properties are reduced to the essential ones. Although it is obvious that a real bank data model is way more complex than the one we propose, we believe that ours is relevant and representative enough and therefore sufficient for the purpose of our work.

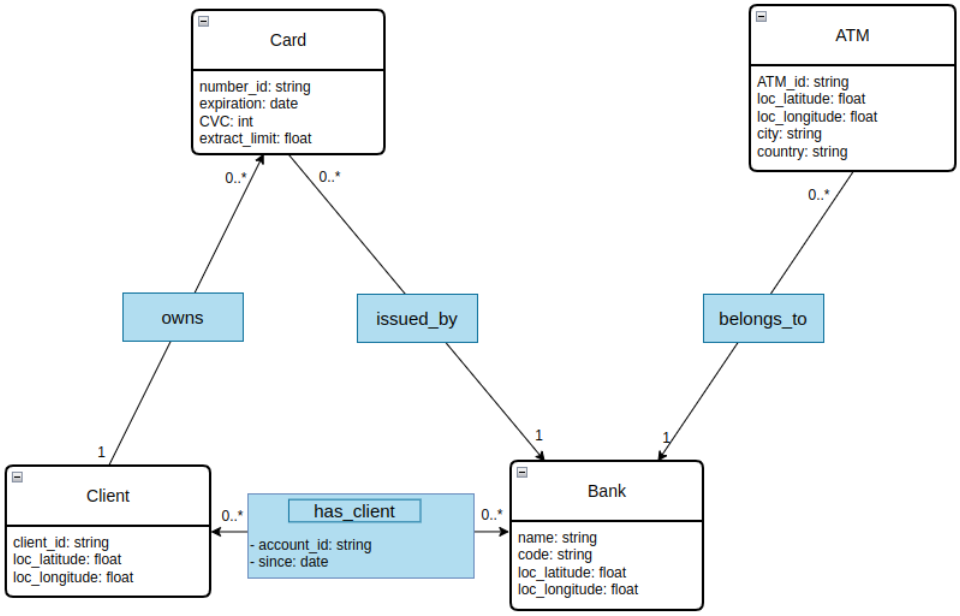

Figure 1: Initial stable property graph model

A first model proposal was the one shown in Figure 7. It contains four entities: Bank, ATM, Client and Cards, and the corresponding relations between them; in particular: a directed relationship from Client to Card: "owns" representing that a client can own multiple credit cards and that a card is owned by a unique client, then a bidirectional relation "has_client" between Client and

Bank; representing bank accounts of the clients in the banks. Then the relation between Card and Bank to represent that a Card is "issued by" a Bank, and that a Bank can have multiple Cards issued. Finally, the relation "belongs_to" between the ATM and Bank entities, representing the ATMs that a Bank owns.

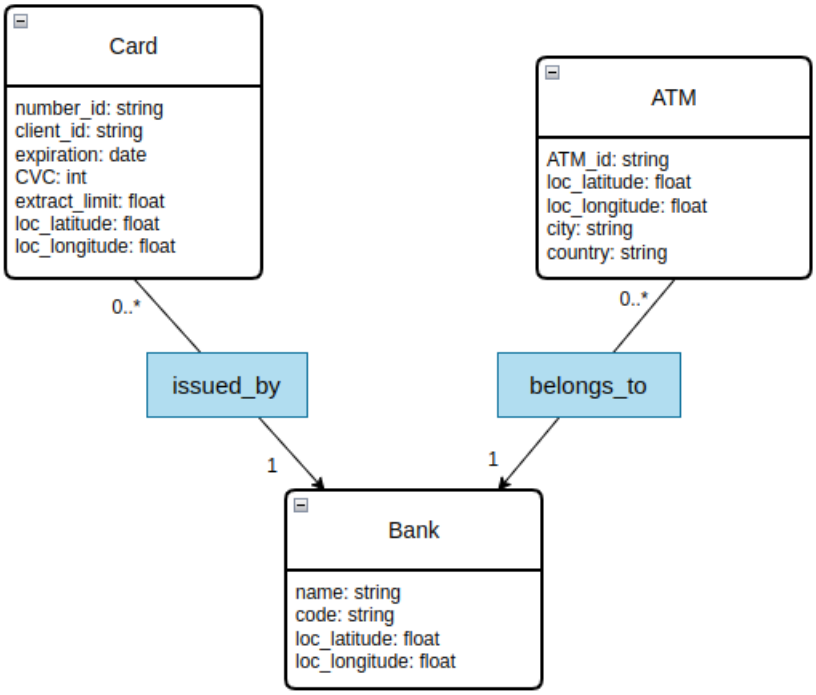

Figure 2: Definitive stable property graph model

However, the final version of the model (see Figure 2) was simplified to reduce it to the minimal needed entities. In particular, Client entity was decided to be removed and included inside the Card entity. This reduction allows to simplify the stable property graph to only three entities. For this, all the Client attributes were included in the Card entity. In the initial Figure 7 schema the Client entity was defined with three attributes: the identifier of the client and the GPS coordinates representing the usual residence of the client. This change is done while preserving the restriction of a Card belong to a unique client the same way it was previously done with the relation between Card and Client "owns" in the initial schema, which now is therefore removed.
Another derived consequence of this simplification is the removal of the other relation that the Client entity had with other entities: the "has_client" relation between Client and Bank, which was originally made with the intention of

representing the bank accounts between clients and banks. Maintaining a bank account would imply having to consistently update the bank account state after each transaction of a client, and, since for the so far considered fraud detection patterns we are not considering patterns related with the accounts, the removal of the bank account relation is negligible and at the same time helpful for the simplification of the model. However, for the sake of completeness the attribute *extract_limit* is introduced in the Card entity, representing a money amount limit a person can extract, which will be related with the amount of money a person owns. This will allow the detection of anomalies related with frequent or very high expenses.

The final entities and their selected attributes are described in what follows:

Entities

Bank

- name: Bank name.
- code: Bank identifier code.
- loc_latitude: Bank headquarters GPS-location latitude.
- loc_longitude: Bank headquarters GPS-location longitude.

Bank	
name	Bank name
code	Bank identifier code
loc_latitude	Bank headquarters GPS-location latitude
loc_longitude	Bank headquarters GPS-location longitude

Figure 3: Bank entity attributes

ATM

- ATM_id: Unique identifier of the ATM.
- loc_latitude: GPS-location latitude where the ATM is located.
- loc_longitude: GPS-location longitude where the ATM is located.
- city: City in which the ATM is located.

- country: Country in which the ATM is located.

ATM	
ATM_id	Unique identifier of the ATM
loc_latitude	GPS-location latitude where the ATM is located
loc_longitude	GPS-location longitude where the ATM is located
city	City in which the ATM is located
country	Country in which the ATM is located

Figure 4: ATM entity attributes

For the moment, this entity is understood as the classic Automated Teller Machine (ATM), however note that this entity could potentially be generalized to a Point Of Sale (POS), allowing a more general kind of transactions apart from the current Card-ATM transactions, where also online transactions could be included apart from the physical ones.

Card

- number_id: Unique identifier of the card.
- client_id: Unique identifier of the client.
- expiration: Validity expiration date of the card.
- CVC: Card Verification Code.
- extract_limit: Limit amount of money extraction associated with the card.
- loc_latitude: Client address GPS-location latitude.
- loc_longitude: Client address GPS-location longitude.

Card	
number_id	Unique identifier of the card
client_id	Unique identifier of the client
expiration	Validity expiration date of the card
CVC	Card Verification Code
extract_limit	Limit amount of money extraction associated with the card
loc_latitude	Client address GPS-location latitude
loc_longitude	Client address GPS-location longitude

Figure 5: Card entity attributes

Note that for both the ATM and the Card entities we have the GPS coordinates information. In the first case referring to the geolocation of each specific ATM and in the last case referring to each specific client address geolocation. This will be useful to be able to detect transaction frauds related to geolocation distances.

Note that the client is completely anonymized in the system (no name, surname, age, or any other confidential details) by using only a client_id. For the present purpose it is enough to uniquely identify each client.

Relations There are only two relations in the final stable subgraph, that is, the ones that are left after the deletion of the Client entity from the first version of the stable subgraph; the relation "issued_by" between the Card and the Bank entities and the "belongs_to" between ATM and Bank entities. For the moment they do not have any attribute, however they could potentially be added in the case this was needed.

0.1.2 Volatile property graph

It contains the minimal needed information to be able to recognize the anomaly fraud patterns we want to identify. This subgraph describes the most volatile part of our model, meaning the transactions between the client's cards and the ATMs. The idea is to have a data model to define the transactions, as continuous temporal interactions between the Card and ATM entities, restricting these kind of relations to the volatile subgraph. The idea is to have a really simple and light property graph schema single-centered on the transactions. For that the Card and ATM entities will be simplified to the last bullet, containing only the identifier of the both entities that each transaction relation matches. These

identifiers will be enough to be able to recover, if needed, the whole information about the specific Card or ATM entity in the stable subgraph.

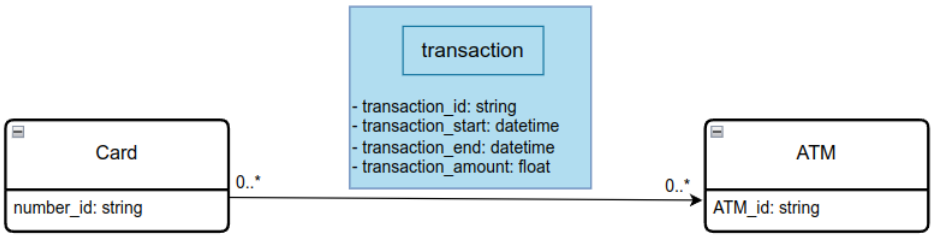

Figure 6: Volatile property graph model

Following this idea of volatile subgraph, the Card and ATM entities and transaction relations of it will not be permanently saved, but instead, only for a certain window of time under the mission of detecting anomalous bank operations.

The transaction relation will include some attributes that will be essential for the detection of the fraud patterns, in particular:

- transaction_id: Unique identifier for each transaction in the database.
- transaction_start: Datetime when the transaction started. Format: DD/MM/YYYY HH:MM (ex. 1/1/2022 4:50).
- transaction_end: Datetime when the transaction was completed. Format: DD/MM/YYYY HH:MM (ex. 1/1/2022 4:54).
- transaction_amount: Amount of money involved in the transaction.

Transaction	
transaction_id	Unique identifier for each transaction in the database
transaction_start	Datetime when the transaction started - format: DD/MM/YYYY HH:MM (ex. 1/1/2022 4:50)
transaction_end	Datetime when the transaction was completed - format: DD/MM/YYYY HH:MM (ex. 1/1/2022 4:54)
transaction_amount	Amount of money involved in the transaction

Figure 7: Transaction relation attributes

0.2 Synthetic dataset creation

Given the confidential and private nature of bank data, it was not possible to find any real bank datasets. In this regard, a synthetic property graph bank dataset was built based on the *Wisabi Bank Dataset*[1]. It is a fictional banking dataset that was made publicly available in the Kaggle platform.

This synthetic bank dataset was considered of interest as a base for the synthetic bank database that we wanted to develop. The interest to use this bank dataset as a base was mainly because of its size: it contains 8819 different customers, 50 different ATM locations and 2143838 transactions records of the different customers during a full year (2022). Additionally, it provides good heterogenity on the different kind of transactions: withdrawals, deposits, balance inquiries and transfers.

The main uses of this bank dataset are the obtention of a geographical distribution for the locations of our generated ATMs and the construction of a card/client *behavior*, for which the data of the *Wisabi Bank Dataset* will be used.

Details of the *Wisabi Bank Dataset* The *Wisabi Bank Dataset* consists on ten CSV tables. Five of them are of transaction records of five different states of Nigeria (Federal Capital Territory, Lagos, Kano, Enugu and Rivers State) that refers to transactions of cardholders in ATMs. In particular they contain 2143838 transactions records done during the year 2022, of which 350251 are in Enugu, 159652 in Federal Capital Territory, 458764 in Kano, 755073 in Lagos and 420098 in Rivers. Then, the rest of the tables are: a customers table ('customers_lookup') where the data of 8819 different cardholders is gathered, an ATM table ('atm_location lookup') with information of each of the 50 different locations of the ATMs, and then three remaining tables as complement of the previous ones ('calendar lookup', 'hour lookup' and 'transaction_type lookup') (tables summary).

In what follows we give the details on the generation of the instances of our static database entities. For simplicity and to do it in a more stepwise manner, we are going to first create all the CSV data tablesfor the nodes and for the relations in the corresponding format and then we will populate the Neo4j GDB with them.

Bank

Since a unique bank instance is considered, the values of the properties of the bank node are manually assigned, leaving them completely customisable. Bank node type properties consist on the bank *name*, its identifier *code* and the location of the bank headquarters, expressed in terms of *latitude* and *longitude* coordinates, as seen in Table 1. For the bank, we will generate `n` ATM and `m` Card entities. Note that apart from the generation of the ATM and Card node types we will also need to generate the relationships between the ATM

[1] Wisabi bank dataset on kaggle

and Bank entities (`belongs_to` and `external`) and the Card and Bank entities (`issued_by`).

Name	Description and value
`name`	Bank name
`code`	Bank identifier code
`loc_latitude`	Bank headquarters GPS-location latitude
`loc_longitude`	Bank headquarters GPS-location longitude

Table 1: Bank node properties

ATM

Name	Description and value
`ATM_id`	ATM unique identifier
`loc_latitude`	ATM GPS-location latitude
`loc_longitude`	ATM GPS-location longitude
`city`	ATM city location
`country`	ATM country location

Table 2: ATM node properties

The bank operates `n` ATMs, categorized in:

- Internal ATMs: ATMs owned and operated by the bank. They are fully integrated within the bank's network.
- External ATMs: These ATMs, while not owned by the bank, are still accessible for the bank customers to perform transactions.

Both types of ATMs are considered to be of the same type of ATM node. Their difference is modeled as their relation with the bank instance: `belongs_to` for the internal ATMs and `external` for the external ATMs, having:

```
n = n_internal + n_external
```

where `n_internal` is the number of internal ATMs owned by the bank and `n_external` is the number of external ATMs that are accesible to the bank.

The ATM node type properties consist on the ATM unique identifier *ATM_id*, its location, expressed in terms of *latitude* and *longitude* coordinates, and the *city* and *country* in which it is located, as seen in Table 2. Note that the last two properties are somehow redundant, considering that location coordinates are already included. In any case both properties are left since their inclusion provide a more human-understandable way to easily realise about the location of the ATMs.

The generation of `n` ATMs for the bank is done following the geographical distribution of the locations of the ATMs in the *Wisabi Bank Dataset*. On this dataset there are 50 ATMs locations distributed along Nigerian cities. Note that for each of these ATMs locations, there can be more than one ATM. However, this is not taken into account and only one ATM per location is assumed for the distribution.
⇒ Put a plot of the distribution of the ATM locations
This distribution of the ATMs matches the relevance of the location in terms of its population, since the number of ATM locations is larger in the most populated Nigerian cities (30% of the ATM locations are in the city of Lagos, then the 20% in Kano...). Therefore, for the generation of the location of each of the `n` ATMs, the location/city of an ATM selected uniformly at random from the *Wisabi Bank Dataset* is assigned as *city* and *country*. Then, new random geolocation coordinates inside a bounding box of this city location are set as the *loc_latitude* and *loc_longitude* exact coordinates of the ATM.
Finally, as the ATM unique identifier *ATM_id* it is assigned a different code depending on the ATM internal or external category:

$$ATM\_id = \begin{cases} bank\_code + " - " + i & 0 \leq i < \texttt{n\_internal} \text{ if internal ATM} \\ EXT + " - " + i & 0 \leq i < \texttt{n\_external} \text{ if external ATM} \end{cases}$$

Card

Name	Description and value
`number_id`	Card unique identifier
`client_id`	Client unique identifier
`expiration`	Card validity expiration date
`CVC`	Card Verification Code
`extract_limit`	Card money amount extraction limit
`loc_latitude`	Client's habitual address GPS-location latitude
`loc_longitude`	Client's habitual address GPS-location longitude

Table 3: Card node properties

- Explicar las properties con la tabla y de la forma que se hizo descriptiva para ATM y Bank.

The bank manages a total of `m` cards. The Card node type properties, as depicted in Table 3, consist on the card unique identifier *number_id*, the associated client unique identifier *client_id*, as well as the coordinates of the associated client habitual residence address *loc_latitude* and *loc_longitude*. Additionally it contains the card validity expiration date *expiration* and the Card Verification Code, *CVC*.

⇒? Finally, it contains the property *extract_limit* which represents the limit on the amount of money it can be extracted with the card on a single extraction/day?

⇒? Include in the card properties the properties related with the gathered behavior for the card: *withdrawal_day*, *transfer_day*, *withdrawal_avg*...

Aspects to explain:

- Extract_limit: explain how and why?

- Card and client identifiers: so far, although for completeness the *client_id* is included in the properties of the Card node type, note that for simplicity it could be ignored, since due to the purposes of our work, a *one-to-one* relationship between card and client is assumed, meaning that each card is uniquely associated with a single client, and that a client can possess only one card. Therefore, the *client_id* is not relevant so far, but is included in case the database model is extended to allow clients have multiple cards or cards belonging to multiple different clients. For each generated Card instance these identifiers are set as:

$$\begin{cases} number\_id = \text{c-}bank\_code\text{-}i \\ client\_id = i \end{cases} \qquad 0 \leq i < \texttt{m}$$

- `Expiration` and `CVC` properties: they are not relevant, could be empty value properties indeed or a same toy value for all the cards. For completeness the same values are given for all the cards: `Expiration` = 2050-01-17, `CVC` = 999.

- Client's habitual address location (`loc_latitude`, `loc_longitude`): two possible options were designed to define the client habitual residence address. In both cases they are random coordinates drawn from a bounding box of a location/city. The difference is on to do the selection of the location/city:

  1. Wisabi customers selection: Take the city/location of the habitual ATM of a random selected *Wisabi* database customer. Note that in the *Wisabi Bank Dataset* customers contain an identifier of their usual ATM, more in particular, the dataset is designed in such a way that customers only perform operations in the same ATM. With this approach, we maintain the geographical distribution of the *Wisabi* customers.
  2. Generated ATMs selection: Take the city/location of a random ATM of the `n` generated ATMs. This method is the one utilized so far.

- ***Behavior***: It contains relevant attributes that will be of special interest when performing the generation of the synthetic transactions of each of the cards. The defined *behavior* parameters are shown in Table 4.

Behavior parameter	**Description**
`amount_avg_withdrawal`	Withdrawal amount mean
`amount_std_withdrawal`	Withdrawal amount standard deviation
`amount_avg_deposit`	Deposit amount mean
`amount_std_deposit`	Deposit amount standard deviation
`amount_avg_transfer`	Transfer amount mean
`amount_std_transfer`	Transfer amount standard deviation
`withdrawal_day`	Average number of withdrawal operations per day
`deposit_day`	Average number of deposit operations per day
`transfer_day`	Average number of transfer operations per day
`inquiry_day`	Average number of inquiry operations per day

Table 4: *Behavior* parameters

For each card, its *behavior* parameters are gathered from the operations history of a randomly selected customer on the *Wisabi Bank Dataset*, from which we can access the operations log of 8819 different customers for one year time interval. On it, there are four different types of operations that a customer can perform: withdrawal, deposit, balance inquiry and transaction. The parameters for the *behavior* gather information about these four different types of operations.

Note that all these *behavior* parameters are added as additional fields of the CSV generated card instances, so, as mentioned, they can later be utilized for the generation of the synthetic transactions.

Another possible way to assign the *behavior* parameters could be the assignation of the same behavior to all of the card instances. However, this method will provide less variability in the generation of the synthetic transactions than the aforementioned method. Nevertheless, other taylored generation methods to generate different *behavior* for each the cards could also be considered to similarly obtain this variability.

- `extract_limit: amount_avg_withdrawal` $* 5$

---

1 Indexing

Useful for ensuring efficient lookups and obtaining a better performance as the database scales.

→ indexes will be created on those properties of the entities on which the lookups are going to be mostly performed; specifically in our case:

- Bank: `code` ?
- ATM: `ATM_id`
- Card: `number_id`

Why on these ones?

→ Basically the volatile relations / transactions only contain this information, which is the minimal information to define the transaction. This is the only information that the engine recieves from a transaction, and it is the one used to retrieve additional information - the complete information details of the ATM and Card nodes on the complete stable bank database. Therefore these parameters/fields (look for the specific correct word on the PG world) are the ones used to retrieve / query the PG.

By indexing or applying a unique constraint on the node properties, queries related to these entities can be optimized, ensuring efficient lookups and better performance as the database scales.

From Neo4j documentation:

> An index is a copy of specified primary data in a Neo4j database, such as nodes, relationships, or properties. The data stored in the index provides an access path to the data in the primary storage and allows users to evaluate query filters more efficiently (and, in some cases, semantically interpret query filters). In short, much like indexes in a book, their function in a Neo4j graph database is to make data retrieval more efficient.

Some references on indexing:

- Search-performance indexes
- The impact of indexes on query performance
- Create, show, and delete indexes

Okay... but before diving deeper...:

To Index or Not to Index?

> When Neo4j creates an index, it creates a redundant copy of the data in the database. Therefore using an index will result in more disk space being utilized, plus slower writes to the disk.
> Therefore, you need to weigh up these factors when deciding which data/properties to index.
> Generally, it's a good idea to create an index when you know there's going to be a lot of data on certain nodes. Also, if you find queries are taking too long to return, adding an index may help.

From another tutorial on indexing in neo4j