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

In the context of our work we could see the data we are considering to be both static and streaming data, as we are considering a bank system application that contains all the information related to it on the cards, clients. . . , and that it is receiving the streaming of transactions that happens on it. More specifically, the static data can be thinked of the classical bank database data, that is, the data a bank typically gathers on its issued cards, clients, accounts, ATMs. . . . Whereas as the streaming data we can consider the transactions the clients of the bank produce with their cards on ATMs, PoS. . . that reach the bank system. Therefore, due to this nature of the data, we consider a *continuously evolving database* paradigm, 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].

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 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 will not 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 entities 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

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 devoloped 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.

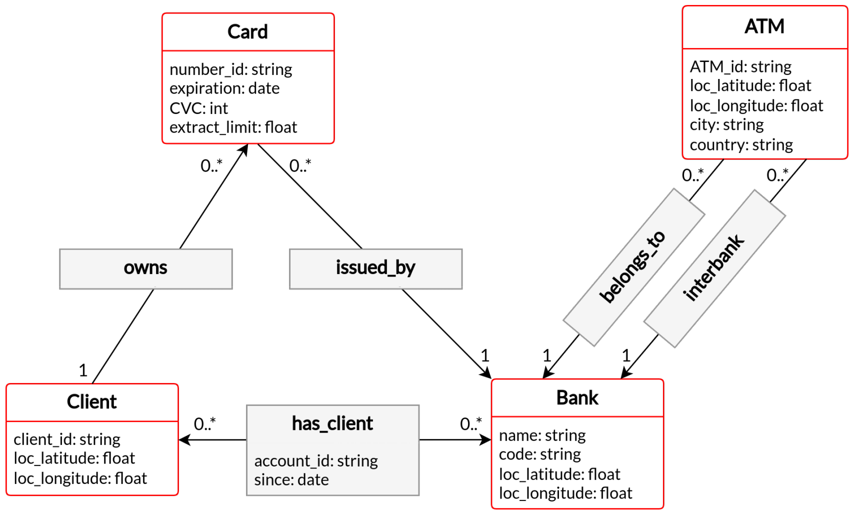

Figure 1: Initial bank stable property graph model

A first model proposal was the one shown in Figure 1. Basically, the idea of this first model attempt was to capture the data that a bank system database typically gathers. It contains four entities: Bank, ATM, Client and Card with their respective properties, and the corresponding relationships between them. The relations are: 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 bank. The relation between Card and Bank to represent that a card is `issued_by` the bank, and that the bank can have multiple cards issued. Finally, the relations `belongs_to` and `interbank` between the ATM and Bank entities, representing the two different kinds of ATMs depending on their relation with the bank; those ATMs owned and operated by the bank and those that, while not owned by the bank, are still accesible for the bank customers to perform transactions.

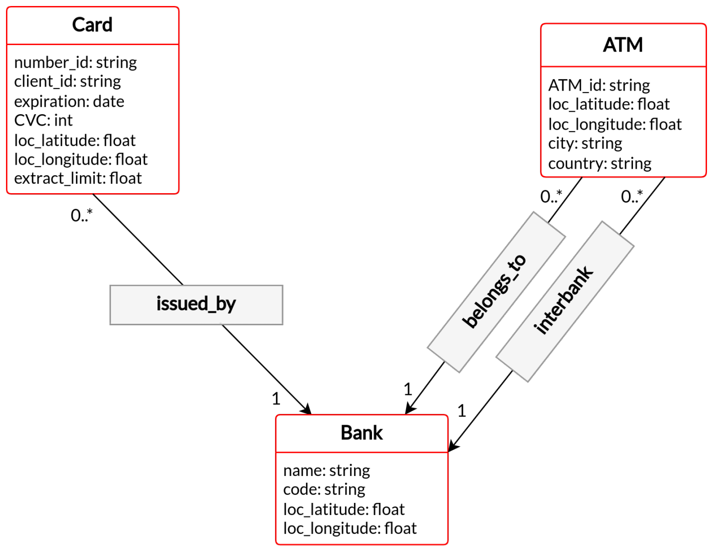

Figure 2: Definitive stable property graph model

However, the final version of the model (see Figure 2) was simplified to reduce it to the minimal needed entities. In particular, we decided to remove the Client entity and to merge it 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 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 belonging 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, complicating the model. Nevertheless, we elimintate the bank account relation, since its removal is considered negligible and at the

same time helpful for the simplification of the model needed for the purposes of our work. 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.

0.2 Synthetic dataset creation

As mentioned, 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.

→ To do the generation of a stable bank database use the Python program `bankDataGenerator.py`, in which you will have to enter the bank properties'

[1]Wisabi bank dataset on kaggle

values, as well as the parameters on the number of the bank ATMs and Cards: `n` and `m`, respectively.

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 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. Modeled with the `belongs_to` relation.
- External ATMs: These ATMs, while not owned by the bank, are still accessible for the bank customers to perform transactions. Modeled with the `interbank` relation.

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:

$$\texttt{n = n\_internal + n\_external}$$

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 maintained since their inclusion provides a more straightforward manner to explore and inspect the created 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:

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

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

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

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*.

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.

⇒? 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*... or just in the CSV to use them for the creation of the synthetic transactions, but do not store them in the stable bank database.

- 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:

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

- `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` $* 5$

0.3 Graph database populating process

0.3.1 Neo4j graph database creation

$\rightarrow$ TODO: 0. Describe on how to set up the database $\rightarrow$ TODO: Explanation of the versions of both Neo4j instances used - local and UPC VM cluster.

Prior to the population of the Neo4j graph database, a Neo4j graph database instance needs to be created. This was done both locally and in a Virtual Machine of the UPC cluster.
Version: Neo4j 5.21.0 Community edition.

- Accessing it: by default it runs on localhost port 7474: `http://localhost:7474`. Start the neo4j service locally by: `sudo systemctl start neo4j`
  It can be also be accessed by the internal utility `cypher-shell`. Username: `neo4j` and password: `bisaurin`.

0.3.2 Neo4j graph database population - CSV to PG

Once we created the Neo4j graph database instance and all the CSV data files representing all the nodes and relations, we populate the Property Graph instance in Neo4j. Before performing the population of the GDB, we create uniqueness constraints on the properties of the nodes that we use as our *de facto* IDs for the ATM and Card IDs: `ATM_id` and `number_id`, respectively. The reason to do this is to avoid having duplicated nodes of these types with the same ID in the database. Therefore, as an example, when adding a new ATM node that has the same `ATM_id` as another ATM already existing in the database, we are aware of this and we do not let this insertion to happen.

ID uniqueness constraints are created with the following cypher directives:

```
CREATE CONSTRAINT ATM_id IF NOT EXISTS
FOR (a:ATM) REQUIRE a.ATM_id IS UNIQUE

CREATE CONSTRAINT number_id IF NOT EXISTS
```

```
FOR (c:Card) REQUIRE c.number_id IS UNIQUE

CREATE CONSTRAINT code IF NOT EXISTS
FOR (b:Bank) REQUIRE b.code IS UNIQUE
```
Listing 1: Uniqueness ID constraints

Once we created the Neo4j graph database instance and all the CSV data files representing all the nodes and relations, we populate the Property Graph instance in Neo4j. For this, we propose two different methods. The first does it by directly importing the CSV files using the Cypher's `LOAD CSV` command, while the second method does it by parsing the CSV data and running the creation of the nodes and relationships using Cypher. Both methods can be found and employed using the `populatemodule` golang module. In this module we can find the two subdirectories where each of the methods can be run. In detail, the module tree structure is depicted in Figure 3. On it, the `cmd` subdirectory contains the scripts to run each of the populating methods: the first method script on `csvimport` and the second on the `cypherimport`, while the `internal` subdirectory is a library of the files with the specific functions used by these methods.

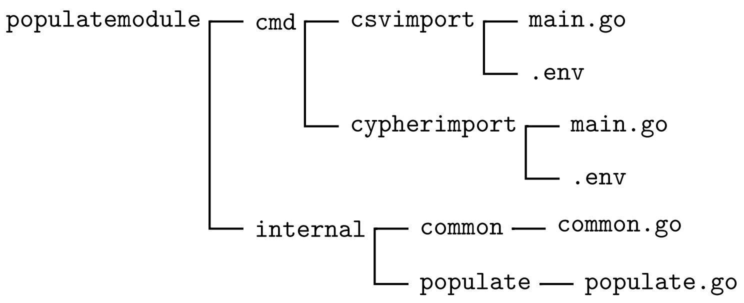

Figure 3: `populatemodule` file structure

Prior to run any of these methods we need to first set up correctly the `.env` file located inside the desired method directory, where we have to define the corresponding Neo4j URI, username and password to access the Neo4j graph database instance.

- **Method 1: Cypher's `LOAD CSV`**
  The Cypher's `LOAD CSV` clause allows to load CSV into Neo4j, creating the nodes and relations expressed on the CSV files (see *load-csv cypher manual*). To use it simply follow these steps:

  1. Place all the CSVs (`atm.csv`, `bank.csv`, `card.csv`, `atm-bank-internal.csv`, `atm-bank-external.csv` and `card-bank.csv`) under the `/var/lib/neo4j/import` directory of the machine containing the Neo4j graph database instance.
  2. Run `$ go run populatemodule/cmd/csvimport/main.go`

Process description: Then the different CSV files containing all the data tables of our data set, were loaded into the GDB with the following cypher directives.

ATM (atm.csv)

```
LOAD CSV WITH HEADERS FROM 'file:///csv/atm.csv' AS row
MERGE (a:ATM {
    ATM_id: row.ATM_id,
    loc_latitude: toFloat(row.loc_latitude),
    loc_longitude: toFloat(row.loc_longitude),
    city: row.city,
    country: row.country
});
```

Listing 2: atm.csv

Some remarks:

- `ATM` is the node label, the rest are the properties of this kind of node.
- Latitude and longitude are stored as float values; note that they could also be stored as cypher *Point* data type. However for the moment it is left like this. In the future it could be converted when querying or directly be set as cypher point data type as property.

Bank (bank.csv)

```
LOAD CSV WITH HEADERS FROM 'file:///csv/bank.csv' AS row
MERGE (b:Bank {
    name: row.name,
    code: row.code,
    loc_latitude: toFloat(row.loc_latitude),
    loc_longitude: toFloat(row.loc_longitude)
});
```

Listing 3: bank.csv

Note that the `code` is stored as a string and not as an integer, since to make it more clear it was already generated as a string code name.

ATM-Bank relationships (atm-bank-internal.csv and atm-bank-external.csv)

```
LOAD CSV WITH HEADERS FROM 'file:///csv/atm-bank-internal.
csv' AS row
MATCH (a:ATM {ATM_id: row.ATM_id})
MATCH (b:Bank {code: row.code})
MERGE (a)-[r:BELONGS_TO]->(b);
```

Listing 4: atm-bank-internal.csv

```
LOAD CSV WITH HEADERS FROM 'file:///csv/atm-bank-external.
csv' AS row
MATCH (a:ATM {ATM_id: row.ATM_id})
MATCH (b:Bank {code: row.code})
MERGE (a)-[r:INTERBANK]->(b);
```

Listing 5: atm-bank-external.csv

Card (card.csv)

```
LOAD CSV WITH HEADERS FROM 'file:///csv/card.csv' AS row
MERGE (c:Card {
    number_id: row.number_id,
    client_id: row.client_id,
    expiration: date(row.expiration),
    CVC: toInteger(row.CVC),
    extract_limit: toFloat(row.extract_limit),
    loc_latitude: toFloat(row.loc_latitude),
    loc_longitude: toFloat(row.loc_longitude)});
```

Listing 6: card.csv

Notes:

- We do not include the fields that were generated to define the behavior of the card. They are only used for the generation of the transactions.
- `expiration`: set as *date* data type.

Card-Bank relationships (card-bank.csv)

```
LOAD CSV WITH HEADERS FROM 'file:///csv/card-bank.csv' AS
row
MATCH (c:Card {number_id: row.number_id})
MATCH (b:Bank {code: row.code})
MERGE (c)-[r:ISSUED_BY]->(b);
```

Listing 7: card-bank.csv

- Method 2:

→ TODO: Describe the other population method

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

→ Apparently, there are *Token lookup indexes* which are a default type of node indexes, from the Neo4j documentation: *"Two token lookup indexes are created by default when creating a Neo4j database (one node label lookup index and one relationship type lookup index). Only one node label and one relationship type lookup index can exist at the same time."*. That is, two indexes are created by default, they are on the node label and on the relationship type.

→ There are different type of indexes depending on what we want to use them for, the type of property that they index...: see here for more details.

→ An example on when to use indexes: example case.

→ **Decision**: Create indexes on the ATM and Card node properties which serve as our node identifiers in practice: on `ATM_id` and on `number_id` node properties. The reason is that, so far, most of the queries are performed looking at a single node using its identifier; in particular to retrieve ATM nodes location coordinates to calculate the distance between two ATM nodes.
POSSIBLE IMPROVEMENT → make Neo4j obtain/calculate/return this distance as the result of a query.

Description of the creation on indexes

1. Select the type of index: range index VS text index. . . Some references on this election:
   - Use range index (the default), if we do not need to use `CONTAINS`, `ENDS WITH` or the value size is not above 8k.
2. Note that for the text index we (may) need to guarantee that the property type is STRING: create a single property type constraint.

1.0.1 Volatile property graph

References

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[2] Rohit Kumar Kaliyar. Graph databases: A survey. In *International Conference on Computing, Communication & Automation*, pages 785–790. IEEE, 2015.