# 1 Synthetic dataset creation

Given the confidential and private nature of bank data, it was not possible to find any real bank datasets to perform our experiments on. 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 designed and architected by Obinna Iheanachor, and that was made publicly available in the Kaggle platform.

In particular it contains 10 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).

Based on the aforementioned dataset we created our own synthetic dataset. For simplicity and to do it in a more stepwise manner, we are going to first create all the CSV data tables for the nodes and for the relations in the corresponding format and then we will populate the Neo4j GDB with those. The description of the creation of the CSV data tables as well as the specific details are described in what follows:

**Bank**

⇒ Only 1 bank

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

The generation of the Banks is done taylored to our needs as we are only considering a reduced amount of Bank entities. In particular we generated three different Banks. `n` ATMs and `m` Cards belonging to each of them, will be generated as explained in what follows. Note that apart from the generation of the entities ATM and Card we will also need to generate the relationships ATM-Bank (`belongs_to`) and Card-Bank (`issued_by`) that matches each of these entities with the corresponding Bank entity to which they belong.

[1] Wisabi bank dataset on kaggle

**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.

The generation of `n` ATMs for the bank is done following the geographical distribution of the locations of the ATMs in the wisabi dataset. On it there are 50 ATMs locations distributed along Nigerian cities. The distribution of the ATMs matches the importance of the location 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 generating a new ATM location first we select uniformly at random an ATM location/city from the wisabi dataset, which is directly assigned as `city` and `country` to the ATM instance, and generate new random geolocation coordinates inside a constructed bounding box of this city location to set as the `loc_latitude` and `loc_longitude` of the ATM.

⇒ Okay like this: Note that we do not take into account for the density distribution of the ATMs of the wisabi dataset that, for each ATM location of the dataset, we have `x` number of atms.
Aspects to explain:

- Geographical distribution of the ATMs based on the geographical distribution of the locations of the Wisabi ATMs.
- Do a plot of the geographical dist of the positions of the ATMs of the wisabi dataset.

**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.

Aspects to explain:

- Extended behavior fields: to not only withdrawal.
- Location: Explain the 2 options we have developed and the one used so far.
- Explain how the behavior of the card is generated based on a random selected wisabi customer.
- Extract_limit: explain how and why?

For the Card entity generation there are some different aspects that are worth to mention:

- First, in relation with the card and client identifiers (`number_id`, `client_id`), for the moment we define that each client has 1 card, later this can be modified.
  $\Rightarrow$ For the moment 1 client : 1 card. So far, for the kind of frauds we are considering this is the easier approach. In a future, if needed, it could be generated the case that 1 client has n cards

- `Expiration` and `CVC` fields: they are not relevant, could be empty fields indeed or for all the Cards the same values. For simplicity and completeness we chose them to be the same values for all the Cards.

  Expiration: set for completeness the same date in all of them but in a far future!

- Behavior: For each Card object to be generated, we assign it a *behavior* based on the transaction behavior of a randomly selected wisabi customer. The behavior is gathered from the transaction history of the customer on the wisabi dataset. This will be particularly useful later for the generation of the synthetic transactions, so that for each of the cards their transactions can be simulated based on this gathered behavior. In particular the customer *behavior* refers to:
  - `extract_limit`: maximum normal amount that a card can extract. `amount_avg` $* 5$.
  - `amount_avg`: extracted amount average of the customer on a transaction.
  - `amount_std`: extracted amount standard deviation of the customer on a transaction.
  - `withdrawal_day`: average number of transactions per day of the customer. Withdrawals only for the moment!.

- New added behavior: interesting for the generation of transactions that are not only withdrawals: balance inquiries, deposits, transfers.

Note that all this fields are additionally added to the Card CSV records. Some additional remarks:

  - For the moment we only consider the *withdrawal* type of transaction in the behavior. However *transfer* and *deposit* could be also considered.
  - This behavior is gathered from one random customer of the wisabi dataset per each of the Cards, so that we have more variability. However, we could also assign the same behavior to all the clients, and this behavior be like a summary of all the wisabi dataset clients behavior. Also the behavior could be assigned drawning it from taylored distributions selected by us, in a more customizable manner.

- Location (`loc_latitude`, `loc_longitude`): Two possible options in this case:
  - 1. Assign a random location of the usual ATM city/location of the random selected wisabi customer. This way, we maintain the geographical distribution of the wisabi customers.
  - 2. Assign a random location of the city/location of a random ATM of the newly generated ATMs objects. (* For the moment)

# 2 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