TFM-FernandoMartín

Fernando Martín Canfrán

December 2, 2024

Edelmira: Ejemplo de comentario de Edelmira. Cada quien debe escoger un color y definir su comando (ver a linea 140)

Edelmira: ATENCIÓN: Por ahora solo hay un editor adicional al owner del proyecto. El departamento no renovó la licencia de overlief porque en *muy breve* tendremos disponibles la licencia institucional...

1 Data Model

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 [**angles2008survey**, **kumar2015graph**].

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.

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 developed our own proposal of a bank database model.

1.1 Design of the Data Model

In what follows we describe the design of our data model as a Property Graph data model, divided into the stable and volatile property graphs. 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. In this regard, we developed our own proposal of a bank database model taking as standard reference the *Wisabi Bank Dataset*[1], which is a fictional banking dataset publicly available in the Kaggle platform.

1.1.1 Stable Property Graph

Taking into account the reference dataset model, a common bank data model could be the one we defined in Figure 1, which intents to capture the data that a bank system database typically gathers.

It contains four entities: Bank, ATM, Client and Card with their respective

[1] https://www.kaggle.com/datasets/obinnaiheanachor/wisabi-bank-dataset

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 accessible for the bank customers to perform transactions.

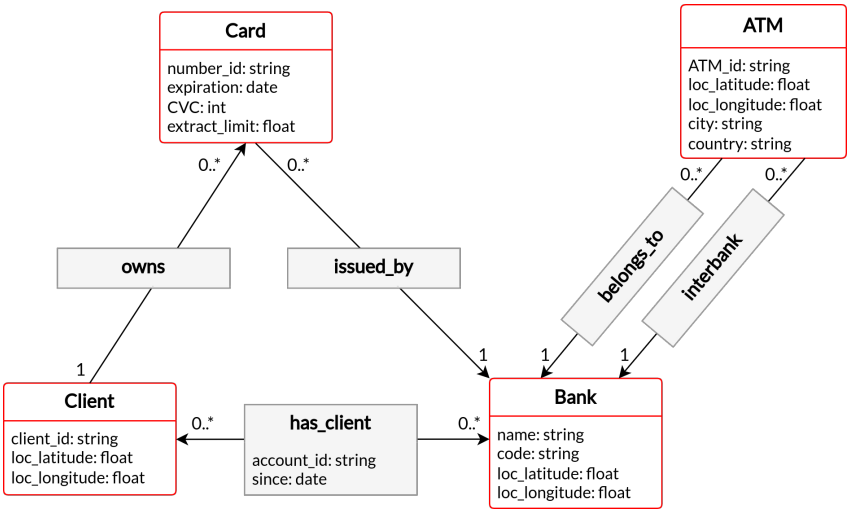


Figure 1: Initial bank stable property graph model

Although this data model is quite close to reality, we chose a simplified version of it as the final model for our proof of concept (see Figure 2). On this final version the defined entities, relations and properties modeling the bank database are reduced to the essential ones, which, we believe are enough to create a relevant and representative bank data model sufficient for the purposes of our work.

On the final version of the model, we decided to remove the Client entity and to merge it inside the Card entity. For this, all the Client properties were included in the Card entity. In the complete data model schema (Figure 1) the Client entity was defined with three properties: 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 complete schema, which now is therefore removed.
Another derived consequence of this simplification is the removal of the other

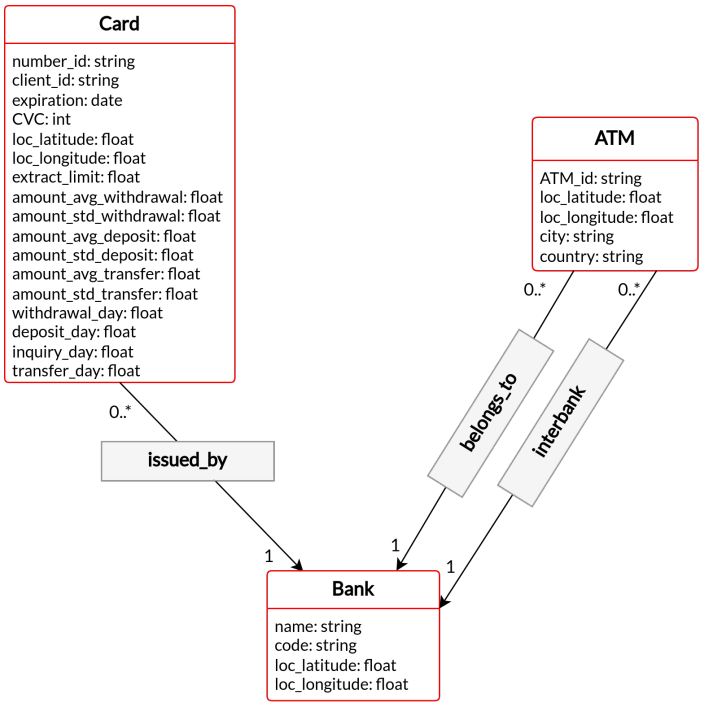


Figure 2: Definitive stable property graph model

relation that the Client entity had with other entities: the `has_client` relation between Client and Bank, which was 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 eliminate 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 property *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. Other properties that are included with the purpose of allowing the detection of some other kinds of anomalies are the GPS coordinates and the client's *behavior* properties. The GPS coordinates are added in the ATM and Card entities, in the first case referring to the geolocation of each specific ATM and in the last case referring to each specific client address geolocation. The client's *behavior* properties are added in the Card entity. They are metrics representing the behavior of the owner of the Card, and they are included as properties as we think they could be of interest to allow the detection of some kinds of anomalies in the future.

As a result, our final simplified stable property graph model contains three node entities: Bank, Card and ATM, and three relations: `issued_by` associating Card entities with the Bank entity, and `belongs_to` and `interbank` associating the ATM entities with the Bank entity.

The Bank entity represents the bank we are considering in our system. Its 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.

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

Note that, from the beginning we were considering more than 1 bank entity. This lead to consider the creation of this entity, which now as only 1 bank is considered it may not be needed anymore, being able to reformulate and simplify the model. However, it is left since we considered it appropiate to be able to model the different kinds of ATMs a bank can have with different relation types instead of with different ATM types.

The ATM entity represents the Automated Teller Machines (ATM) that either belong to the bank's network or that the bank can interact with. For the moment, this entity is understood as the classic ATM, however note that this entity could potentially be generalized to a Point Of Sale (POS), allowing a more general kind of interactions apart from the current Card-ATM interaction, where also online transactions could be included apart from the physical ones. We distinguish two different kinds of ATMs, depending on their relation with the bank:

- 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 `interbank` for the external ATMs.

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 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 explicit description of the location of the ATMs.

Finally, the Card node type represents the cards of the clients in the bank system. 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*, the coordinates of the associated client habitual residence address *loc_latitude* and *loc_longitude*, the card validity expiration date *expiration*, the Card Verification Code, *CVC* and the *extract_limit* property, which represents the limit on the amount of money it can be extracted with the card on a single withdrawal. Finally it contains the properties related with the *behavior* of the client: *amount_avg_withdrawal*, *amount_std_withdrawal*, *amount_avg_deposit*, *amount_std_deposit*, *amount_avg_transfer*, *amount_std_transfer*, *withdrawal_day*, *deposit_day*, *transfer_day* and *inquiry_day*.

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
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 3: Card node properties

The client is completely anonymized in the system (no name, surname, age, or any other confidential details) by using only a *client_id*. Currently, *client_id* is included in the Card node type for completeness. However, it could be omitted for simplicity, as we assume a one-to-one relationship between card and client for the purposes of our work – each card is uniquely associated with a single client, and each client holds only one card. Thus, the *client_id* is not essential at this stage but is retained in case the database model is expanded to support clients with multiple cards or cards shared among different clients.

1.1.2 Volatile Property Graph

The volatile subgraph model describes the most *variable* part of our model, the continuous interactions between the client's cards and the ATMs. These interactions are continuously occurring and arrive to our system as a continuous data stream. This subgraph model contains the minimal information needed to identify the Card and ATM entities – `number_id` and `ATM_id` Card and ATM identifiers – between which the interaction occurs, along with additional details related to the interaction. The proposed data model can be seen in the Figure 3. On it we define the Card and ATM node entities with only the identifier properties, `number_id` and `ATM_id`, respectively. These identifiers are enough to be able to recover, if needed, the whole information about the specific Card or ATM entity in the stable subgraph. Finally we define the *interaction* relationship between the Card and the ATM nodes. The *interaction* relation contains as properties: `id` as the interaction unique identifier, `type` which describes the type

of the interaction (withdrawal, deposit, balance inquiry or transfer), `amount` describing the amount of money involved in the interaction in the local currency considered, and finally, `start` and `end` which define the interaction *datetime* start and end moments, respectively.

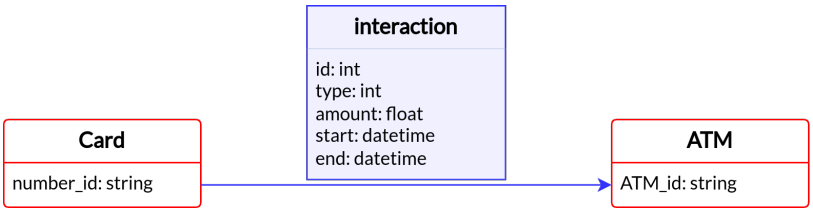


Figure 3: Volatile property graph model

As a whole, the proposed property graph data model is represented in Figure 4. where both the stable and volatile property subgraphs are merged to give a full view on the final property graph model.

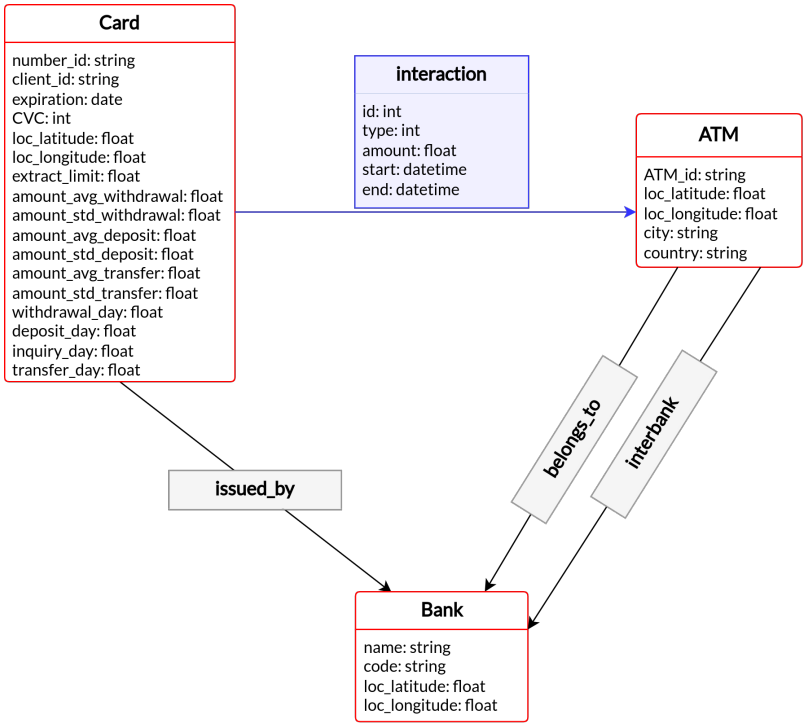


Figure 4: Complete property graph model

1.2 Creation of the Synthetic Dataset

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 stable property graph bank database and a set of synthetic transactions were built based on the *Wisabi Bank Dataset*[2]. The *Wisabi Bank Dataset* is a fictional banking dataset that was made publicly available in the Kaggle platform. We considered it 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*, which we will use for the generation of the synthetic transactions.

In particular, we divide the creation of the synthetic dataset in two. On the one hand on the creation of the stable bank database and on the other hand on the creation of the set of synthetic transactions and anomalous transactions that will conform the stream of data reaching our system.

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).

1.2.1 Stable Bank Database

To do the generation of a stable bank database we provide the Python program `bankDataGenerator.py`, in which it is needed to enter the bank properties' values, as well as the parameters on the number of the bank ATMs (internal and external) and Cards: `n` and `m`, respectively.

How to use the bank data generator

1. Ensure to have the provided `wisabi` directory with the csv files from *Wisabi Bank Dataset*.

[2]Wisabi bank dataset on kaggle

2. Run `$> python3.10 bankDataGenerator.py` – Run with Python3.6 version or higher – Introduce:
   (a) Bank properties' values.
   (b) $n = |ATM|$, internal and external.
   (c) $m = |Cards|$.

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 tables for 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. 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`).

ATM We generate `n = n_internal + n_external` ATMs, 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 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 We generate a total of `m` cards that the bank manages, for each of them the assignment of the different properties is done as follows:

- Card and client identifiers:

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

- `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 how the selection of the location/city is done:

  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 transactions record of a randomly selected customer on the *Wisabi Bank Dataset*, from which we can access the transactions record 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` $* 5$ Other possible ways could be chosen for assigning a value to this property.

1.2.2 Transactions Set

The transaction set constitutes the simulated input data stream continuously arriving to the system. Each transaction represents the operation done by a client's card on a ATM of the bank network. Therefore it has the form of a *interaction* edge/relation from the volatile subgraph (see 1.1.2) matching one Card with one ATM from the stable bank database.

Note that, as in our definition of the input data stream of the DP_{CQE}, we will generate two edges per transaction – the *opening* and the *closing* edge – which both will constitute a single *interaction* relation. The *opening* edge (Figure 5) will be the indicator of the beginning of a new transaction between the matched Card and ATM, it contains the values of the properties related with the starting time `start`, the transaction `type` as well as the `id`. The *closing* edge (Figure 6) will indicate the end of the transaction, completing the values of the rest of the properties of the *interaction*: `end` and `amount`.

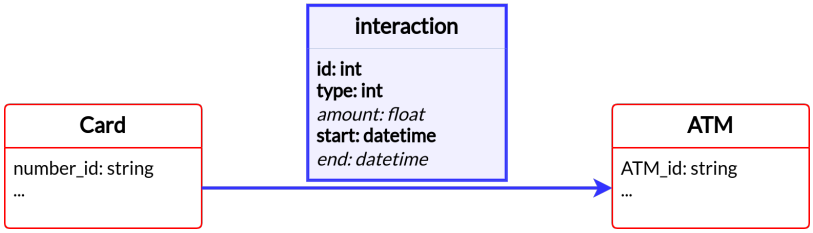


Figure 5: *Opening* interaction edge

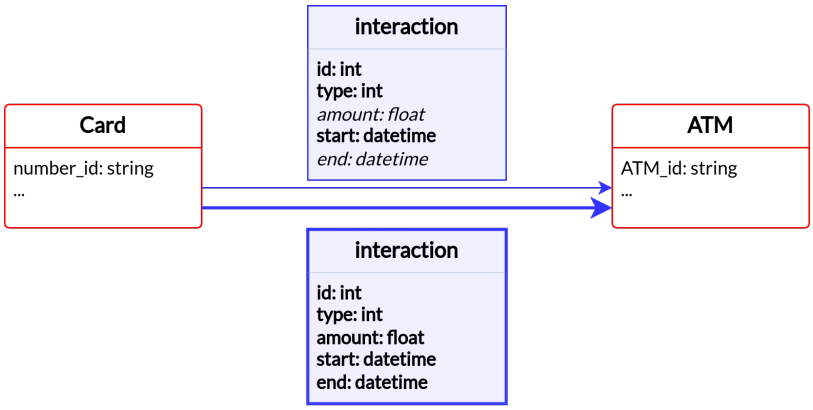


Figure 6: *Closing* interaction edge

We divide the generation of the transaction set in two subsets: the regular transaction set and the anomalous transaction set. The regular transaction set consists of the *ordinary/correct* transactions, whereas the anomalous transaction set is composed of the *irregular/anomalous* transactions that are intentionally created to produce anomalous scenarios. The main objective under this division is the ability to have the control on the exact number of anomalous ATM transactions that we generate so that we can later measure the efficiency of our system detecting them. as we have the exact number of anomalous transactions created to compare with, represented by size of the anomalous subset. Otherwise, if we were generating all the transactions together at the same time it would be more difficult to have the control on the amount of anomalous scenarios created, so that later, it will not be possible to measure the efficiency of our system detecting them, since we will not know their amount.

To do the generation of the synthetic set of transactions we created the Python program `transactionGenerator.py`. On it we need to specify the value of the parameters needed to customise the generation of the set of transactions.

Notas de Fernando... → para eliminar

- ⇒**By card**: Generation of the transactions for each of the cards independently. **We have control** to avoid anomalous scenarios when selecting the ATMs and distributing the transactions along time.
- **In general**: Linking ATM and client composing (card,ATM) pairs, and distributing these pairs along time according to a certain distribution. **No control / More difficult to control the possible derived anomalous scenarios produced among same card pairs**.

Therefore, the generation by card option it is considered to be the best so to be able to have the control on the possible anomalous scenarios for the generated transactions of each of the cards. Some ideas to explore:

- Selection of ATMs:
  - ⇒ Neighborhood / Closed ATM subset.
  - Random walk. To do the selection of the sequence of ATMs for the generated transactions.
- Distribution of the transactions along time:
  - ⇒ Uniform distribution.
  - ⇒ (Consider the possibility) Poisson process distribution.
- Other options:
  - Random walk for both the ATM and the transaction time selection, in the same algorithm together.

TODOS:

- Cambiar/Actualizar dibujos
- Poner lista de params y explicar (tabla) como y qué se puede configurar
- Poner instrucciones de como correr el generador!!!
- Prerequisitos: csv directory has to exist - create it beforehand better
- Output files that are generated: regular, anomalous and all csvs.
- Anomalous generator:
  - NO-Overlapping assumption - Explain
  - Any type of tx to produce the fraud -¿ does not matter the type for the FP1.

Regular Transaction Set

The key idea of the transactions of this set is to avoid the creation of anomalous scenarios among them, so to have a close-to-reality simulation of a bank transaction stream flow free of all the kinds of anomalies considered. After this set is created, the transactions producing anomalous scenarios related with each specific fraud pattern will be produced.

We generate transactions for each of the generated cards on our bank network, based on each of the gathered card transaction behavior. The regular transaction data stream is generated for a customisable `NUM_DAYS` number of days starting in a `START_DATE`. As a summary of the procedure we first show a pseudocode of the transaction generator in Algorithm 1, of which later some of its parts are explained.

Algorithm 1 Regular Transactions Generation

```
1: id ← 0
2: for card in cards do
3:    ATM_subset, ATM_subset (complement) ← createATMsubset(residence_loc)
4:    t_min_subset ← calculate_t_min_subset(ATM_subset)
5:    num_tx ← decide_num_tx()
6:    T ← distribute(num_tx, t_min_subset)
7:    for t_i in T do
8:       ATM_i ∼ ATM_subset
9:       start_i ← t_i.start
10:      end_i ← t_i.end
11:      type_i ← getType()
12:      amount_i ← getAmount()
13:      id_i ← id; id ← id + 1
14:      createTransaction(id_i, ATM_i, start_i, end_i, type_i, amount_i)
15:   end for
16:   introduceAnomalous(ATM_subset, ATM_subset (complement))
17: end for
```

1. **Creation of the ATM subset**

   `ATM_subset`, $\overline{\texttt{ATM\_subset}}$ ← createATMsubset(`residence_loc`)

   Among all the ATMs of the stable bank database we create a subset of ATMs `ATM_subset`, such that all the regular transactions generated for this card take place in (randomly selected) ATMs of this subset. `ATM_subset` = {`ATM` | `distance(ATM,residence_loc)` ≤ `MAX_DISTANCE_SUBSET_THRESHOLD`}. `ATM_subset` consists of the ATMs that are considered to be *usual* for the card, considering the ATMs that are at a distance lower or equal to a customisable maximum distance threshold `MAX_DISTANCE_SUBSET_THRESHOLD` to the registered residence location `residence_loc` on the card. We also limit the size of this subset, considering only a maximum ratio of the total number of ATMs (`MAX_SIZE_ATM_SUBSET_RATIO` ∈ [0, 1]), so that only a certain ratio of the

closest ATMs are included on it:

$$|\texttt{ATM\_subset}| = \texttt{MAX\_SIZE\_ATM\_SUBSET\_RATIO} * |\texttt{ATM}|$$

TODO: CAMBIAR ESTA IMAGEN

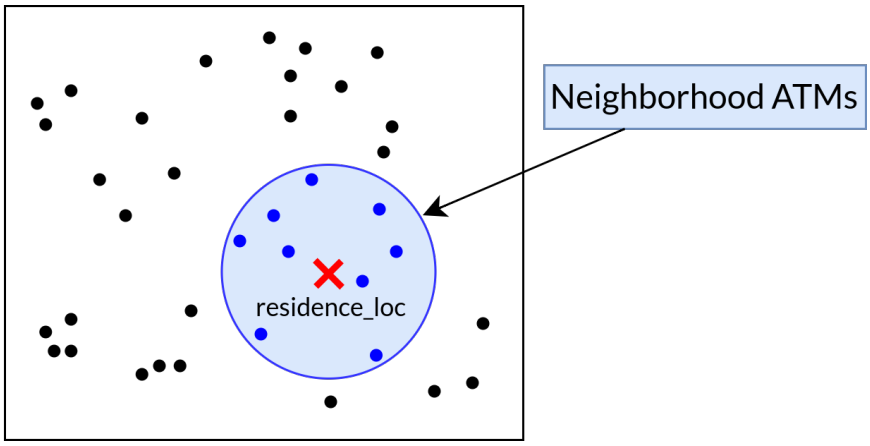


Figure 7: `Neighborhood` ATM subset

2. **Calculate `t_min_subset`**
   `t_min_subset` ← calculate_t_min_subset(`ATM_subset`)

   Minimum threshold time between any two consecutive transactions of the card. That is, the minimum time distance between the end of a transaction and the start of the next consecutive transaction of a card. For it, we take the time needed to traverse the maximum distance between any pair of ATMs of the `ATM_subset`: `max_distance_subset` at an assumed speed that any two points can be traveled for the case of ordinary scenarios: `REGULAR_SPEED`.

$$\texttt{t\_min\_subset} = \frac{\texttt{max\_distance\_subset}}{\texttt{REGULAR\_SPEED}}$$

3. **Decide the number of transactions `num_tx`**
   `num_tx` ← decide_num_tx()

   Based on the behavior of the card, we decide the number of transactions (`num_tx`) to generate for the card for the selected number of days `NUM_DAYS` as:

$$\texttt{num\_tx} \sim \text{Poisson}(\lambda = \texttt{ops\_day} * \texttt{NUM\_DAYS})$$

   where `ops_day` is the sum of the average number of all the kinds of operations per day of the particular card:

$$\texttt{ops\_day} = \texttt{withdrawal\_day} + \texttt{deposit\_day} + \texttt{inquiry\_day} + \texttt{transfer\_day}$$

4. **Decide on the type of transaction**

   $\texttt{type}_i \leftarrow \text{getType}()$

   For each of the `num_tx` transactions, the transaction `type` is decided randomly assigning a transaction `type` given a probability distribution constructed from the card behavior:

$$\begin{cases} P(\texttt{type} = \texttt{withdrawal}) = \frac{\texttt{withdrawal\_day}}{\texttt{ops\_day}} \\ P(\texttt{type} = \texttt{deposit}) = \frac{\texttt{deposit\_day}}{\texttt{ops\_day}} \\ P(\texttt{type} = \texttt{inquiry}) = \frac{\texttt{inquiry\_day}}{\texttt{ops\_day}} \\ P(\texttt{type} = \texttt{transfer}) = \frac{\texttt{transfer\_day}}{\texttt{ops\_day}} \end{cases}$$

5. **Distribution of the `num_tx` transaction times**

   $T \leftarrow \text{distribute}(\texttt{num\_tx}, \texttt{t\_min\_subset})$

   Along the selected time interval starting in `START_DATE` and finishing `NUM_DAYS` days after we do a random uniform distribution of the `num_tx` transaction times. T contains the list of all the the `start` and `end` times tuples for each of the `num_tx` transactions, with the constraint that, for each transaction i, the next one $i+1$ is at a minimum time distance of `t_min_subset`. Specifically, the transaction times are generated guaranteeing:

$$i.\texttt{end} + \texttt{t\_min\_subset} < (i+1).\texttt{start} \ \forall i \in [1, \texttt{num\_tx})$$

   The `end` time of a transaction is assigned a shifted time difference with respect to the `start` time. In particular:

$$\texttt{end} = \texttt{start} + \texttt{time\_difference}$$

   where:

$$\texttt{time\_difference} \sim \mathcal{N}(\texttt{MEAN\_DURATION}, \texttt{STD\_DURATION})$$

   with the corrections:

$$\texttt{time\_difference} = \begin{cases} \texttt{MEAN\_DURATION} & \text{if } \texttt{time\_difference} < 0 \\ \texttt{MAX\_DURATION} & \text{if } \texttt{time\_difference} > \texttt{MAX\_DURATION} \\ \texttt{time\_difference} & \text{otherwise} \end{cases}$$

   TODO: PONER UN DIBUJITO!, explicar lo del checking de los fitting holes? –¿ yo creo que esto ya no es necesario... demasiado detalle

6. Assign a transaction `amount`
   `amount`$_i \leftarrow$ getAmount()

   Amount is assigned depending on the `type` based on card behavior:

$$\begin{cases} \mathcal{N}(\texttt{amount\_avg\_withdrawal}, \texttt{amount\_std\_withdrawal}) & \text{if } \texttt{type} = \texttt{withdrawal} \\ \mathcal{N}(\texttt{amount\_avg\_deposit}, \texttt{amount\_std\_deposit}) & \text{if } \texttt{type} = \texttt{deposit} \\ 0 & \text{if } \texttt{type} = \texttt{inquiry} \\ \mathcal{N}(\texttt{amount\_avg\_deposit}, \texttt{amount\_std\_transfer}) & \text{if } \texttt{type} = \texttt{transfer} \end{cases}$$

$$\text{If } \texttt{amount} < 0, \text{ then re-draw from } U(0, 2 \cdot \texttt{amount\_avg\_type}).$$

   with `amount_avg_type` as `amount_avg_withdrawal`, `amount_avg_deposit` or `amount_avg_deposit` depending on the respective transaction `type`.

Anomalous Transaction Set

After the generation of regular transactions we perform an injection of transactions to produce anomalous scenarios. The injection is taylored depending on the specific kind of anomalous scenarios that we want to produce. In what follows we explain the injection process depending on each of the types of frauds that we have considered.

Fraud Pattern I To produce anomalous scenarios related to this type of fraud, we produce the injection of transactions that will produce the satisfaction of this fraud pattern. In other words, we inject transactions that violate the minimum *time-distance* constraint between transactions performed with the same card. Therefore, as we can see in Figure 8, if we consider a set of regular transacions for a certain card, where y_1 and y_2 are regular consecutive transactions, we will introduce an anomalous transaction a_{12} such that:

$$(y_1.\texttt{ATM\_id} \neq a_{12}.\texttt{ATM\_id}) \wedge (a_{12}.\texttt{start} - y_1.\texttt{end} < T_{min}(y_1.\texttt{ATM\_loc}, a_{12}.\texttt{ATM\_loc}))$$

where `ATM_loc` is the tuple of coordinates (loc_latitude,loc_longitude) of the corresponding ATM. This injection will produce an anomalous scenario of this kind of fraud with at least the y_1 previous transaction. Note that, it could possibly trigger more anomalous fraud scenarios with the subsequent transactions (y_2 and on...).

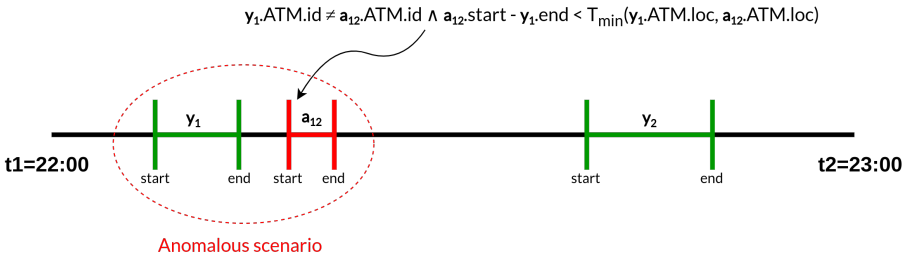


Figure 8: Creation of anomalous scenario - type I

Some assumptions related with the generation of anomalous transactions for this kind of fraud pattern are:

- **Overlapping of transactions is not possible**: Appart from guaranteeing that this injection causes at least one anomalous scenario, we also respect the additional constraint of ensuring that the anomalous transaction injected does not cause overlapping with any of the transactions, in particular neither with the previous nor the next one. This constraint is added based on the assumption that the bank itself does not allow to open a transaction whenever another one is still open. Therefore considering that a_{12} is the anomalous injected transaction in between the regular consecutive transactions y_1 and y_2, when generating a_{12} we guarantee that:

$$\begin{cases} a_{12}.\texttt{start} > y_1.\texttt{end} \\ a_{12}.\texttt{end} < y_2.\texttt{start} \end{cases}$$

- **There are no two consecutive anomalous transactions**: For simplicity in our practical purposes, we do the generation of anomalous transactions for this kind of fraud pattern assuming that an anomalous transaction can only be in between two regular consecutive transactions, so that we do not consider the case of the injection of two or more consecutive anomalous transactions for this kind of fraud. See Figure 9.

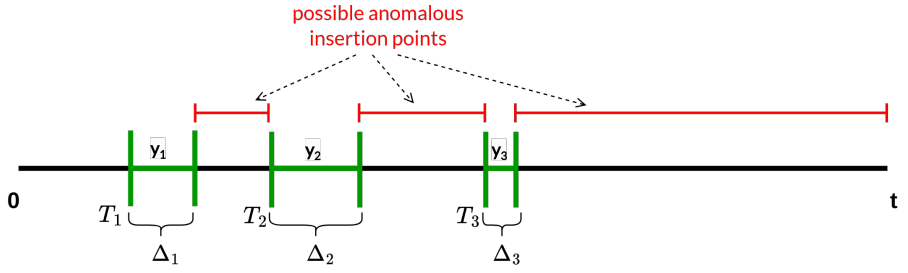


Figure 9: Considered possible injection points of anomalous transactions of fraud type I

We generate ANOMALOUS_RATIO_1 $*$ num_tx anomalous transactions for each of the cards related with the fraud pattern I, where ANOMALOUS_RATIO_1 $\in [0, 1]$ defines the ratio of anomalous transactions of this kind over the total number of regular transactions num_tx for each of the cards. In Algorithm 2 we describe at a high level the process of the generation of anomalous transactions for this kind of fraud pattern.

Preconditions:

- ATMsubset, and the compl. are given by param, from the tx regular generator

Algorithm 2 Introduction of Anomalous Transactions for Fraud Pattern I

introduceAnomalous(ATM_subset, $\overline{\text{ATM\_subset}}$)

```
num_anomalous ← num_tx * ANOMALOUS_RATIO_1
i ← 0
while i < num_anomalous do
    ATM_i ~ ATM_subset (complement)
    prev_i, next_i ← randomUniquePosition(num_tx)
    t_i ← getTime(prev_i, next_i)
    start_i ← t_i.start
    end_i ← t_i.end
    type_i ← getRandomType()
    amount_i ← getAmount()
    id_i ← id; id ← id + 1
    createTransaction(id_i, ATM_i, start_i, end_i, type_i, amount_i)
    i ← i + 1
end while
```

1. **Assignment of ATMs not belonging to the** ATM_subset: the anomalous transactions are linked to ATMs that are part of the complementary of the ATM_subset.

2. **Each anomalous transaction has a unique insertion position**: As described previously, we do not allow the case of two or more consecutive anomalous transactions injection. Each anomalous transaction occupies a unique position among all the possible injection positions defined by the set of regular transactions generated for the card. As it can be seen on Figure 9, considering that we have three regular transactions, we will consider three unique possible insertion points for the anomalous transactions. The procedure of assigning a unique insertion position for each anomalous transaction to be generated is achieved with the function randomUniquePosition(num_tx), that given the number of regular transactions of the card num_tx returns the previous and the next regular transaction to the assigned unique position.

3. **Assign transaction times such that respecting the needed time constraints**: in particular there are two time constraints to be satisfied:

- Production of fraud pattern with prev_i
- No overlapping with prev_i nor with next_i

This is summarized in the pseudocode as the procedure getTime(prev_i, next_i), which returns t_i, as the tuple of (`start`,`end`) times.

4. **Random transaction type**
5. **Arbitrary amount**

1.3 Population of the Graph Database

1.3.1 Neo4j graph database creation

→ TODO: 0. Describe on how to set up the database → 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`.

1.3.2 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 10. 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.

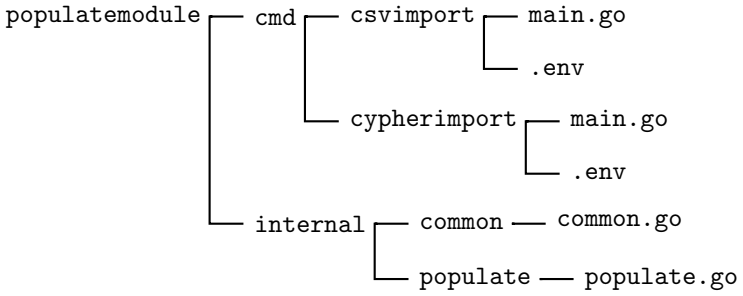


Figure 10: `populatemodule` file structure

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),
  amount_avg_withdrawal: toFloat(row.amount_avg_withdrawal),
  amount_std_withdrawal: toFloat(row.amount_std_withdrawal),
  withdrawal_day: toFloat(row.withdrawal_day),
  amount_avg_deposit: toFloat(row.amount_avg_deposit),
  amount_std_deposit: toFloat(row.amount_std_deposit),
  deposit_day: toFloat(row.deposit_day),
  inquiry_day: toFloat(row.inquiry_day),
  amount_avg_transfer: toFloat(row.amount_avg_transfer),
  amount_std_transfer: toFloat(row.amount_std_transfer),
  transfer_day: toFloat(row.transfer_day)
  });
```

Listing 6: card.csv

Notes:

- We include the fields that were generated to define the behavior of the card. They are also 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: Creation of Cypher queries**

  → TODO: Describe the other population method

  → TODO: Describe the details of the VM of the UPC cluster where the gdb is hosted – in TFM-Neo4j/size-estimaºtion, the gmail and other files...

1.4 Connection to GDB

TODO: Add this in the data model section under a new subsection?

Some details / notes on how this is performed in golang.
So far:

- `DriverWithContext` object: only 1, shared among all the threads. It allows connections and creation of sessions. These objects are immutable, thread-safe, and fairly expensive to create, so your application should only create one instance.

- `Sessions`: so far we create one session every time we do a `checkFraud()` operation. Session creation is a lightweight operation, so sessions can be created and destroyed without significant cost. Always close sessions when you are done with them. They are not thread safe: you can share the main DriverWithContext object across threads, but make sure each routine creates its own sessions.

- `context`: context variable is not unique, and we will create one different just before needing to call functions related with the connection module.

CHANGED: 1 session per filter.
However, note that many multiple parallel sessions may cause overhead to the database... ... WE ARE GOING TO ASK THE ADMIN TO KNOW THIS...
In the case this is a problem we will need to think of the pool of connections.
Some notes on this:

- Bolt thread pool configuration: Since each active transaction will borrow a thread from the pool until the transaction is closed, it is basically the minimum and maximum active transaction at any given time that determine the values for pool configuration options:
  - `server.bolt.thread_pool_min_size`: 5 by default.
  - `server.bolt.thread_pool_max_size`: 400 by default.

1.5 Neo4j Details - VM Neo4j

Tenemos una VM con Neo4j con 4 cores y 20GB de RAM.

TODO: Add details on the Neo4j gdb and in the specific details of the VM of the UPC cluster in which we have our Neo4j gdb instance.
→ DETAILS ON directory: "TFM/NEO4J" TFM-NEO4J

→En su día nos instalasteis una MV con Neo4j. Hemos hecho algunas pruebas y de momento bien! Sin embargo quería preguntaros acerca de cuál es el límite en el número de sesiones que pueden haber en paralelo (vamos a tener varios procesos en paralelo y cada uno con una sesión abierta para hacer queries a la base de datos. Estas queries en principio son todas de lectura), para entonces dependiendo de esto, saber si esto nos limita a la hora de ajustar el número de procesos que vamos a tener en paralelo. He encontrado alguna referencia aquí: https://neo4j.com/docs/operations-manual/current/performance/bolt-thread-pool-configuration/ donde indican que el número máximo de transacciones activas en un momento dado por defecto está en 400... No sé si esto influye en el número de sesiones o no.
→Pues he estado buscando información y no veo en ningún sitio donde el Neo4j limite el número de sesiones que pueda haber en paralelo. Sí que he encontrado que limita el número de transacciones paralelas a 1000 por defecto, pero nada más.
→Lo mejor es que prepares algunos tests y lo pruebes empíricamente ;)

2 The Query Model

TODO: Formal description of the qeury model

2.1 Fraud Pattern I - Card Cloning

A first algorithmic proposal to detect this kind of fraud pattern is the one shown in the algorithm 3. Note that S refers to the filter's subgraph and e_{new} is the new incoming edge belonging to the filter, such that it is a opening interaction edge, since in the case it is a closing interaction edge, we do not perform the CheckFraud().

Algorithm 3 CheckFraud(S, e_{new}) – **initial version**

Require: S is the subgraph of edges of the filter (sorted by time)
Require: e_{new} is the new incoming opening interaction edge belonging to the filter

```
1:  fraudIndicator ← False
2:  i ← |S|
3:  while i > 0 and fraudIndicator = False do
4:      e_i ← S[i]
5:      t_min ← obtain_t_min(e_i, e_new)
6:      t_diff ← e_new.start − e_i.end
7:      if t_diff < t_min then
8:          createAlert(e_i, e_new)
9:          fraudIndicator ← True
10:     end if
11:     i ← i − 1
12: end while
```

There are some aspects and decisions of this algorithm that are worth to describe:

- **Pairwise detection**. The checking of the anomalous fraud scenario is done doing the check between the new incoming edge e_{new} and each of the edges e_i of the filter's subgraph S.
- **Backwards order checking**. The pairs (e_{new}, e_i) are checked in a backwards traversal order of the edge list of the subgraph S, starting with the most recent edge of the subgraph and ending with the oldest.
- **Stop the checking whenever the first anomalous scenario is detected**. Whenever an anomalous scenario corresponding to a pair (e_{new}, e_i), then we stop the checking at this point and emit the corresponding alert. Therefore we do not continue the checking with previous edges of S.
- **Emission of the pair (e_{new}, e_i) as the alert**. The alert is composed by the pair (e_{new}, e_i) that is detected to cause the anomalous scenario. Both edges are emitted in the alert since we do not know which is the one

that is the anomalous. On the one hand, it can be e_i, which is previous to e_{new}, in the case that e_i at the moment it arrived it did not cause any alert with the previous edges/transactions of the subgraph and it causes it now with a new incoming edge e_{new} which is a regular transaction of the client. On the other hand, it can be e_{new}, which is the last having arrived to the system, that it directly causes the alert with the last (ordinary) transaction of the card.

However, a more detailed study, lead us to a simplification of the initially proposed algorithm to the one shown in 4. On it we just perform the checking between the new incoming edge e_{new} and the most recent edge of the subgraph S, e_{last}.

Algorithm 4 CheckFraud(S, e_{new}) – **definitive version**

Require: S is the subgraph of edges of the filter (sorted by time)
Require: e_{new} is the new incoming opening interaction edge belonging to the filter
1: $last \leftarrow |S|$
2: $e_{last} \leftarrow S[last]$
3: `t_min` $\leftarrow$ obtain_t_min(e_{last}, e_{new})
4: `t_diff` $\leftarrow e_{new}.start - e_{last}.end$
5: **if** `t_diff` $<$ `t_min` **then**
6: createAlert(e_{last}, e_{new})
7: **end if**

In what follows we argument the reason why it is sufficient to just check the fraud scenario among e_{new} and the last/most recent edge of the subgraph and not have to continue having to traverse the full list of edges.

Assume that we have a subgraph as depicted in Figure 11, and that we do not know if there have been anomalous scenarios produced between previous pairs of edges of the subgraph. Name $F_I(y_i, y_j)$ a boolean function that is able to say whether it exists an anomalous fraud scenario of this type between the pair of edges (y_i, y_j) or not. In addition, note that the edges of the subgraph S are ordered by time in ascending order, in such a way that $y_1 < y_2 < y_3$. Finally note that $y_3 \equiv e_{new}$ as it is the new incoming edge and $y_2 \equiv e_{last}$, since it is the last edge / the most recent edge of S.

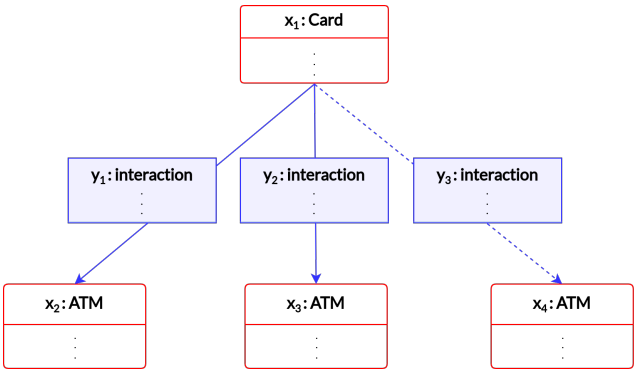


Figure 11: Subgraph S of a card – Fraud Pattern I

Note that we can have that:

- $F_I(y_2, y_3)$: We emit an alert of this anomalous scenario produced between the pair (y_2, y_3). We could continue checking for anomalous scenarios between y_3 and previous edges of the subgraph. However, what we consider important for the bank system is to detect the occurrence of an anomalous scenario in a certain card. Therefore, we consider that, to achieve this, it is enough to emit a single alert of anomalous scenario on this card, and not many related with the same incoming transaction of the same card.

- $\neg F_I(y_2, y_3)$: We analyze whether it would be interesting or not to continue the checking with previous edges of the subgraph, based on assumptions on the fraud checking between previous edges. In particular we can have two cases:

  - If $F_I(y_1, y_2)$: Having this it can happen that either $F_I(y_1, y_3)$ or $\neg F_I(y_1, y_3)$. In the case of $F_I(y_1, y_3)$, since $\neg F_I(y_2, y_3)$, we can infer that the anomalous scenario detected between y_1 and y_3 is a continuation of the same previous anomalous scenario detected between y_1 and y_2. Therefore, we can conclude that this does not constitute a new anomalous scenario that would require an alert.
  - If $\neg F_I(y_1, y_2)$: It can be shown that *by transitivity*, having $\neg F_I(y_1, y_2) \wedge \neg F_I(y_2, y_3) \implies \neg F_I(y_1, y_3)$.
    TODO: Show a formal demostration of this case!

Therefore, we have seen that, it is enough to perform the checking between the pair formed by e_{new} and the most recent edge of the subgraph e_{last}. □

TODO: Explain that we use this proof as a way to show that we do not need to store the full list of edges in the case of this fraud pattern (just the last edge). Maybe for others we need to store more / a list of edges.

TODO: Complete other aspects of the filter worker algorithmic workflow

Others – not so much related with the CheckFraud algorithm, but in general with the filter's algorithm –:

- Save all the edges in the subgraph S, even though they are the reason of the creation of an anomalous scenario.
- Number of anomalous fraud scenarios that can be detected. Bounded by:

$$\#TX\_ANOM \leq SCENARIOS \leq 2 * \#TX\_ANOM$$

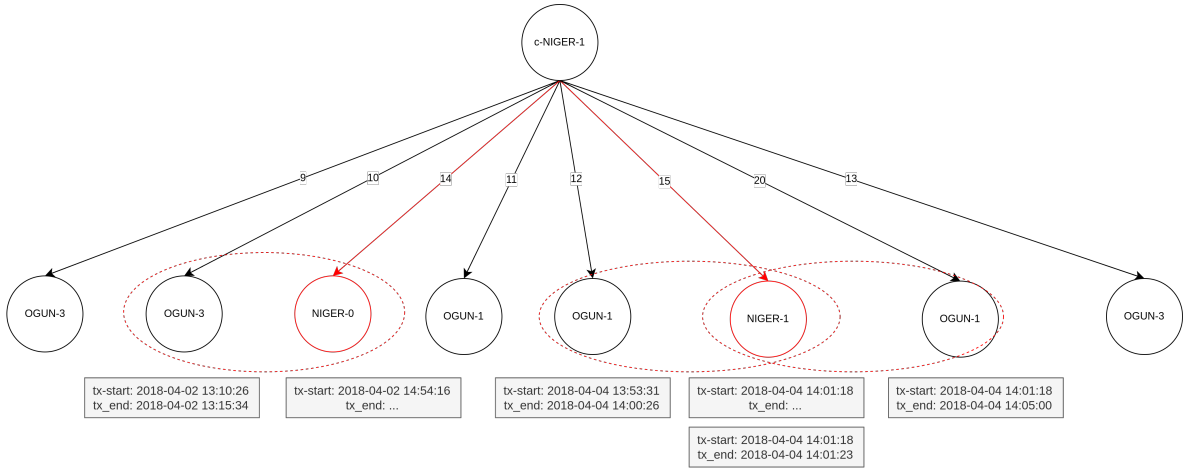
TODO: Poner dibujo y explicar mejor

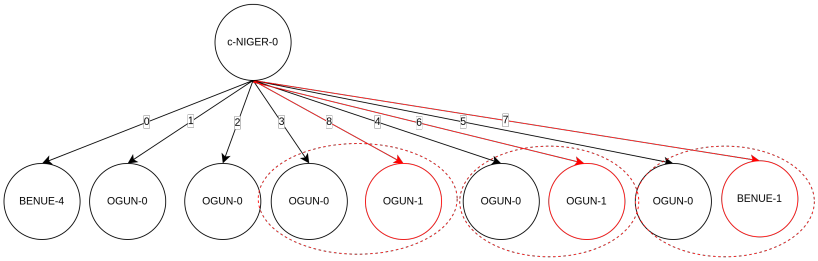
2.2 Initial Tests

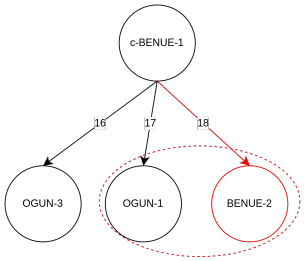
In the following we describe the preliminar tests that were done to check the correct expected behavior of the FP_1 algorithm.

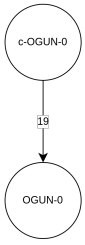
2.2.1 FP_1 - Test 1

2.2.2 FP_1 - Test 1.1









3 Dynamic pipeline

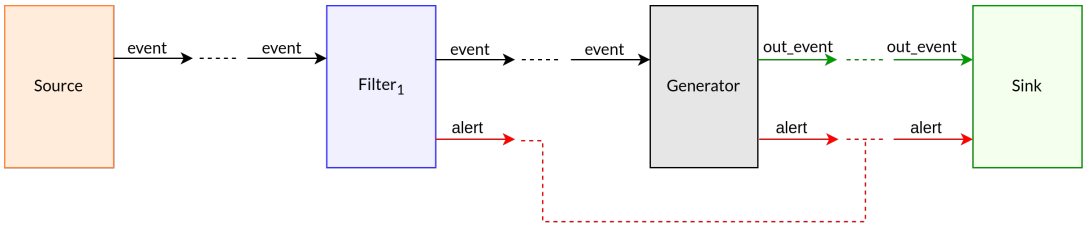


Figure 12: Pipeline Schema

TODO: Switch the description, edge and event channel merged in 1 single channel

Description of the channels:

- `event`: events channel. TODO: Describe the type of events!
- `alert`: direct channel from the filters (in particular the filter worker) to the sink (it does not go through the Generator, although it has it to be able to give it to the filters so that they are able to write on it)
- `out_event`: direct dedicated event channel between Generator and Sink.
- `internal_edge`: edge channel between filter and its worker. Used to communicate to the worker the edges belonging to the filter that the worker needs to process. → Now events and not only edges, and also distinguishing between start and end edges on the type of event.
- `endchan`: synchronization channel between Filter and Worker, to let Filter know whenever Worker is done. To avoid finishing the filter before the worker is actually done. TODO: Include in the drawing

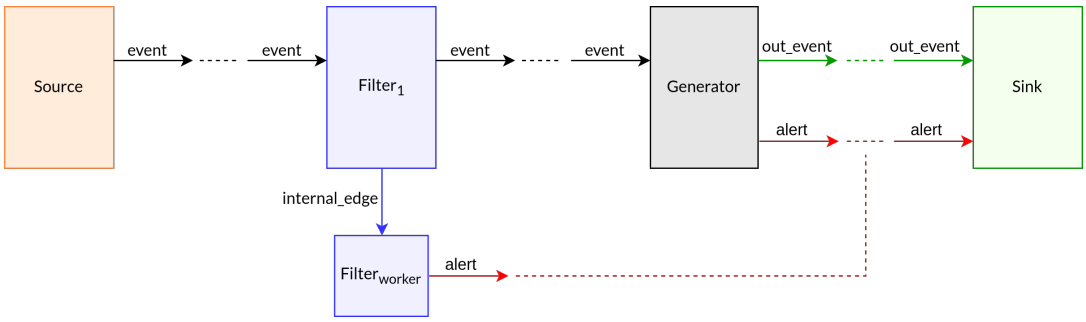


Figure 13: Pipeline Schema with Filter detail

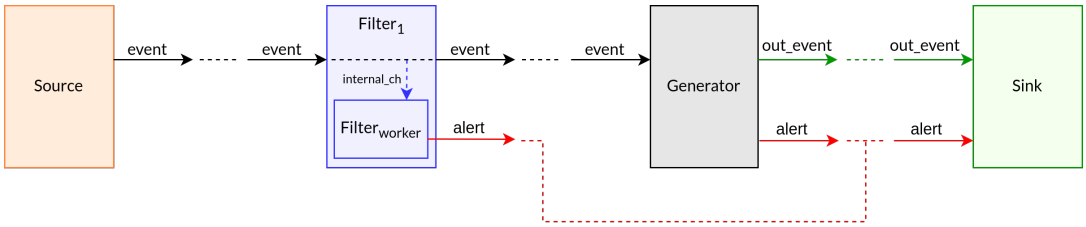


Figure 14: Pipeline Schema with Filter detail 1

Problem detected
If the EOF event is sent through the event channel then it can happen that this event reaches / is treated before the full stream of edges is fully read / processed, leading to the termination of the processes before the processing of all the edges.
Therefore, we decide to merge the edge and the event channel in one single channel!

Some notes on the implementation decisions:

3.1 Filter worker

Options:

- External (named) goroutine
- → Internal anonymous goroutine

Advantages of this decision:

- Code simplification, the filter worker can access the variables of the scope of the filter (no need to pass them as parameters). This is particularly useful in the case of the `alert` channel, to which the worker is able to write directly. Same in the case of the `internal_edge` channel.

and in the case of passing the edges of the card from the filter to the filter worker:

- Shared buffer using mutex
- → Channel

In the case of having a shared buffer to communicate the edges between the filter and the worker a mutex is needed. This is because the filter and the worker can possible write and read, respectively, into this buffer at the same time. With it we will avoid race conditions in the sharing of the buffer. However, a channel or other kind of tool would be needed to indicate the worker that there is an edge ready to be read in the buffer. Not having this, would imply to continuosly have the worker requesting the mutex to read from the buffer, even when it is empty and there is no edge to read.

Therefore as a much more simple alternative, we decided to use an internal channel `internal_edge` in between the filter and the worker. With it we avoid having to use a mutex and leading with its derived coordination issues. As a general use case channels are typically used for *passing the ownership of data* which is the case we are dealing with.

Some links:

- When should you use a mutex over a channel?
- Go Wiki - use a mutex or channel?

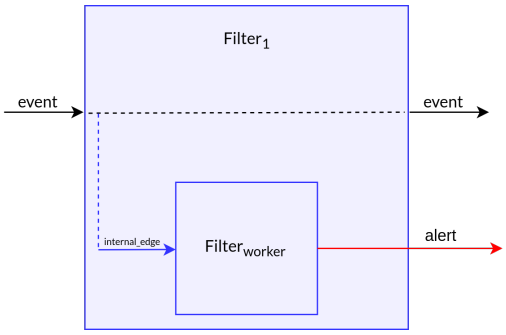


Figure 15: Filter Worker detail

4 Multiple cards per filter support

Use a hash table to index card ids to card subgraphs

- key: card id
- value: pointer to Graph

Note that golang maps are inherently dynamic in size. -¿ control the desired maximum size by ourselves.

Management:

- Filter
  - Reads to check the existance of an entry on the map
  - Creates the entries on the map (in the corresponding cases)
- Worker
  - Modifies the entries on the map once they are created (Filter does not modify values after the creation of the entry)

Conclusion: Safe to do it with a single map and without mutex, since there can not be concurrent writes on the same map entries. Filter writes on the creation and then it is only the worker who writes on that entry after the creation of the entry by the filter.

Issue: "fatal error: concurrent map read and map write"

More details – in Golang it is not defined what happens when we have simultaneous read/write operations:

- Maps Atomicity
- Runtime. Use of maps
- Golang maps. Concurrency
- Golang maps concurrency - blog

4.1 Solution

2 hash tables (to avoid race conditions in concurrent access by filter & worker):

- `cardList`: to control the belonging cards to the filter. Only access by filter.
- `cardSubgraph`: to map each belonging card to its corresponding subgraph. Only access by worker.

Note that, all the edges that we pass to the worker are of cards that belong to the corresponding filter. This means that we do not need to do any check on the `cardSubgraph` map.

TODO: Pendiente de explicar – revisar "dynamicPipeline.tex

- Algorithms of filter, filterWorker, and the rest of stages.
- Completion of tx when arriving closing interaction edge.
- Data structure description of volatile subgraph.
- Description that anomalous tx are also stored in the subgraph.

TODO: Reformulate this description – single window VS windowed approach

4.1.1 Structure description

In principle, we are going to dedicate one filter per card. A filter will be tracking the activity of a card during a certain active time period, by filtering all the transactions belonging to that card and keeping a *windowed* volatile subgraph that contains the transactions of the filter associated card during the last fixed window of time. Once a certain time period has passed without registering any

activity on the filter, that is, there are no transactions associated with the card it *hosts*, the filter will be destroyed and the pipeline reconnected accordingly. Note that the deletion of the filters after a certain time of inactivity is done due to the huge overhead that it would imply to keep them all simultaneously whenever we are considering a big amount of cards.

4.1.2 Transaction flux

4.1.3 Filter's management and volatile subgraph

With the purpose of tracking the activity of each of the cards on each of the filters we build a *windowed* volatile subgraph, which has as edges all the transactions belonging to the card that were registered during the last fixed window of time. This volatile subgraph is going to be updated based on the flux of transactions of the pipeline. We need to have this volatile continuously updating subgraph per each of the cards as a *short-term memory* register of the last transactions of a card during a certain window of time, so that whenever a new transaction comes we can make associations/relations with the transactions registered on that last window of time and possibly alert of a potential fraud whenever it is the case.
Therefore, the continuous update consists of adding new incoming transactions and discarding the old transactions that fall outside the fixed window of time. This ensures that only transactions inside the fixed window of time are considered to do associations for fraud detection. Based on these premises, the data structure to keep the volatile subgraph needs to have two main properties:

- Efficient to add new transactions/edges.
- Efficient to delete the transactions/edges that are outdated in relation to the fixed *window* of time.

Considering that the transactions arrive ordered in time, a linked list was selected as a suitable approach to save the transactions/edges ordered by timestamp. It allows a cheap and easy way to manage the volatile subgraph by adding the new transactions at the tail of the list and deleting the outdated ones from the head of the list. The golang `container/list`, implemented as a doubly linked list was used as an already given implementation of the desired linked list.

Another possible alternative considered was the golang `[] slice` which is a dynamically-sized array. The problem of it was the inefficiency on the deletion from the head of the slice, which had a time complexity of $O(n)$, where n is the number of elements in the slice, since it required shifting all the remaining elements to the left.

Another aspect to consider is the deletion of the filter whenever no transactions related to it are registered after a long period of time. Note that the deletion of the filters after a certain time of inactivity is done due to the huge overhead that it would imply to keep them all simultaneously whenever we are considering a big amount of cards.

The filter's data structure and the operations related to it are described in what follows.

Data Structure

```
type Graph struct {
        last_timestamp time.Time
        edges          *list.List
}
```

Listing 8: filter subgraph data structure

The filter inner data structure `Graph` contains the linked list needed to save the volatile subgraph: `edges`, which is a linked list of transactions (`Edge`) belonging to the filter within the fixed window of time (see an example in Figure 16), and a timestamp: `last_timestamp` which saves the timestamp of the last edge that was added to the filter's subgraph. This is used to check for the deletion of the filter in the case of a long time of *inactivity*.

```
// It is an edge of the volatile subgraph
type Edge struct {
        Number_id string    // Card id
        ATM_id    string    // ATM id
        Tx_id     int64     // transaction id
        Tx_start  time.Time // transaction start date time (DD/MM/
            YYYY HH:MM:SS)
        Tx_end    time.Time // transaction end date time (DD/MM/
            YYYY HH:MM:SS)
        Tx_amount float32   // transaction amount
}
```

Listing 9: Edge of the volatile subgraph, a transaction belonging to the filter

Figure 16: Subgraph `edges` linked list of transactions of the `c-NIGER-0` card filter example

Operations The operations related with the inner filter data structure are the following:

- `NewGraph()`: Creates a new `Graph` data structure. This is done on the creation of the filter.

- `AddAtEnd(e Edge)`: Adds a new edge. Appends a new edge at the end of the list `edges` and updates the `last_timestamp` variable with the `Tx_end` time of the new added edge. This operation is done whenever a new transaction edge of the associated card arrives to the filter.

- `Update(timestamp time.Time)`: Given a certain datetime, it updates the subgraph, starting from the first edge of the `edges` list, by eliminating those that are outdated with respect to this datetime. An edge e is outdated whenever its time difference with respect to the incoming timestamp is greater than the fixed window time constant `timeTxThreshold`. That is, whenever:

  $$timestamp - e.Tx\_end \geq timeTxThreshold$$

  This operation is done in two situations:

  - $edge \in filter$ ($filter.Number\_id = edge.Number\_id$) Whenever the transaction edge belongs to our filter. Before performing the `AddAtEnd` operation we perform the `Update` operation with the timestamp of the new transaction. This ensures that only transactions inside the fixed window of time are considered to do associations for fraud detection.
  - $edge \notin filter$ ($filter.Number\_id \neq edge.Number\_id$) Whenever a transaction edge passes through our filter but it does not belong to it. We take its timestamp as the `timestamp` parameter of the `Update` operation.

- `CheckFilterTimeout(timestamp time.Time) bool`: Given a timestamp, it tests if the filter has to be deleted due to a long time of detected *inactivity*. A filter is decided to be deleted if the time difference between the last edge of the subgraph (saved in the `last_timestamp` variable) and the incoming `timestamp` is greater than the fixed `timeFilterThreshold` time constant:

  $$timestamp - last\_timestamp \geq timeFilterThreshold$$

  This operation is only performed whenever $edge \notin filter$, taking its timestamp as our parameter, in particular before the `Update` operation, since in the case the filter needs to be deleted, the `Update` operation will no longer have to be performed.

- `CheckFraud(new_e Edge) bool`: TODO: This is the temporal approximate approach So far, it is assumed that the current volatile subgraph of the filter is correct (in the sense that it is considered to be free of anomalous transactions). For the moment, only the kind of fraud explained at section **??** is checked. More specifically, the way to perform the check is ONLY between the new incoming edge belonging to the filter `new_e` and the last edge that was added to the volatile subgraph. And if, this kind

of fraud pattern is matched then an alert is output and this last edge is considered to be anomalous and therefore it is not added to the volatile subgraph of the filter.

With these operations, a summary of the algorithmic behavior of a filter can be seen in the flow diagram on Figure 17.

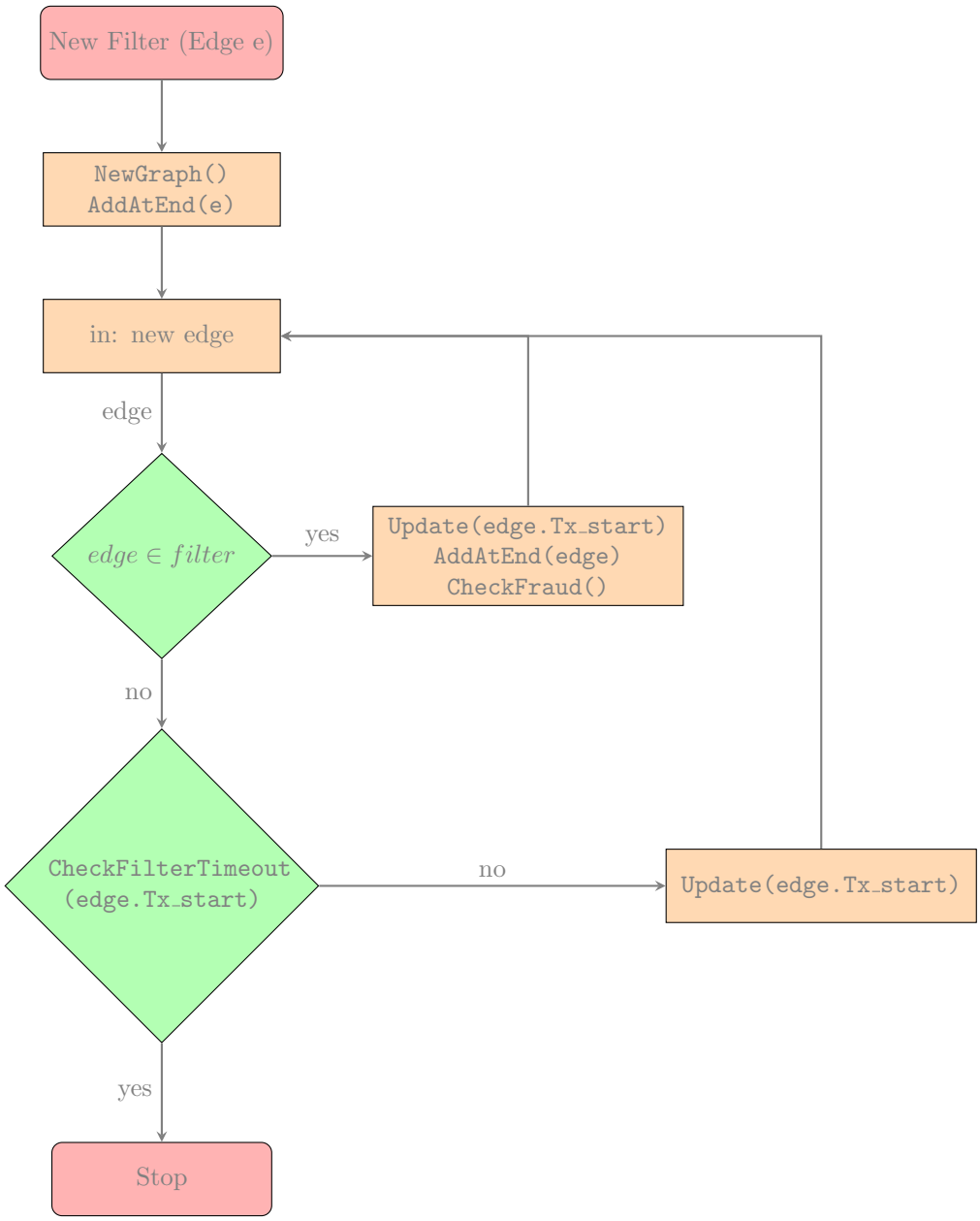


Figure 17: Filter flow diagram

Considerations

- So far, the way to perform the `Update` and the `CheckFilterTimeout` op-

erations could be described as a kind of *filter lazy updating*, since only the filters located before the filter to which the transaction edge finally belongs are updated with the timestamp of that edge. Whereas the filters located after this filter will not be updated with this timestamp. The cause of this is that once the edge is filtered to its corresponding filter, it *sinks* into it, and therefore it is not propagated to the consecutive filters. See the case on Figure 18, where if a transaction edge was to belong to the filter F3, then only the filters F1, F2 and F3 will be updated accordingly to the timestamp of this edge, whereas the filter F4 will not be updated in this case.

Figure 18: Pipeline *filter lazy updating* example

However, this is not necessarily a problem. The reason is that the filter will perform these operations later in the future. Note that in the case that the edge belongs to the filter, the `Update` operation will be performed just before doing the `AddAtEnd` operation as it was already mentioned in the description of the `Update` operation. Note that to avoid this "problem" another approach could be done like doing a timestamp channel or passing all the edges until the end of the pipeline.

- Due to the nature of the management of the filter's lifetime and subgraph updating, it can be the case that our filter subgraph is empty, although the filter is still alive. We can arrive to this situation by having deleted all the edges of the filter subgraph since they were outdated, but still the `timeFilterThreshold` time has not yet passed. However, thanks to the saved `last_timestamp` variable we do not need to save the last edge or do anything but comparing the incoming timestamp with this `last_timestamp` variable (calling the function `CheckFilterTimeout()`).

5 Experiments

5.1 Experimental Approaches

Note that, because of the way we did the transaction generator (coming from wisabi database client's behavior), the average number of transactions per day per card is ~ 1, and therefore to be able to generate a transaction set with anomalous situations more close to reality, a reasonable time interval size for the generated transaction stream would be having T around some weeks or month(s).

5.1.1 1st option: Real time-event stream simulation

Since we do not have the material time to run each experiment for a interval time T of some weeks or a month the idea is to do time scaling of the time event stream. We take the stream of a certain time interval size T and map it into a smaller time interval T' where $T' << T$. Then, we do a real-time event simulation, providing the events of the input stream to the system at the times they actually occur (in reality possibly with a small certain delay!) using their timestamps.

- **Shorter experimental time**: Reduced time to test the system behavior. Instead of T, only T' time to test it.
- **Stress testing - Graph database size - amount of filters' subgraphs**: We do not test the system under a real-case scenario considering its number of cards c, instead we are testing it under a higher load to what it would correspond, but having c cards, and therefore c filter's subgraph. The benefit is that we do not need to have such a big graph database.

The consequences for the experiments and metrics:

- **Diefficiency metrics** (continuous delivery of results): If we give the input stream to the system respecting the temporal timestamps, note that no matter the system characteristics, that a result (an alert in our case), will not be possible to be produced until the event causing it arrives to the system. Therefore the emission of events is expected to be really similar in this case, for any system variation. Only in the case when the stream load is high enough we expect to see some differences?? → HABRÁ QUE IR VIÉNDOLO...
- **Response time**: having in mind the previous considerations, we think in measuring the possible differences of behavior of the different system capabilities in terms of the mean response time. The mean response time (`mrt`) would be the average time that the system spends since it receives the transactions involved in an alert until the time it emits the alert.

Problems derived to pay attention to:

- Shrinking the timestamps to a smaller time interval, produces the emergence of not real fraud patterns that before did not exist due to their real and "correct" larger time distance. Example:
  - Consider the original size of the time interval of the input stream $T = 120h$ (5 days) and $T' = 24h$.
  - Consider two consecutive regular transactions of a certain client performed in two different ATMs `ATM-x` and `ATM-y` with `t_min`= 8h (minimum time difference to traverse the distance from `ATM-x` to `ATM-y`) and `t_diff`= 24h (time difference between the first and the second transaction).
  - → Note that with the scaling the time difference `t_diff` would be of 5 times less, that is, `t_diff` $= 4.8h$. Therefore this will make `t_diff'` $= 4.8h <$ `t_min` $= 8h$.
- → (*) Solution A: **introduce the scaling factor as a input parameter** and consider it also for the fraud checking so to properly **scale the `t_min` variable** (`t_min` $= 8h \rightarrow$ `t_min'` $= \frac{8}{5}h = 1.6h$) and therefore:
  - Before scaling: `t_diff` $= 24h >$ `t_min` $= 8h$.
  - After scaling (scale factor $= \frac{1}{5}$): `t_diff` $= 24 * \frac{1}{5} = 4.8h >$ `t_min` $= 8 * \frac{1}{5} = 1.6h$.
- → Solution B: conserve the original timestamps, and consider the mapped-reduced timestamps for simulating the arrival times of the transactions into the system while taking the original timestamps for the checking of the frauds.

5.1.2 2nd option: real timestamp omission

Do not consider the real-time simulation, by omitting the transaction timestamps in the sense that we do not consider them to simulate a real case scenario where each transaction arrives to the system at the time indicated by its timestamp. Instead all the stream comes (ordered by timestamp) but directly (almost) at the same time to the system. With this approach:

- **No real case simulation**
- **Measure the load the system can take**: for the different system variations given a same stream.
- **Diefficiency metrics**: since time arrival of the transactions to the system is now ignored, and all the transactions come one after the other, a result to be produced do not need to wait for the real timestamp of the transaction. Therefore, we could see the differences in continuously delivering results of the different systems under the same input stream load (more clear than before).

Some (other) references:

- Apache Flink: distributed processing engine for stateful computation of data streams.

5.2 Experiments' Objective

5.2.1 Approach 1: Real time simulation

IMPORTANT: WHAT DO WE WANT TO TEST?
Definition of the objectives of the experiments:

- See and compare the behavior of the system(s) with different streams (different number of cards, greater or smaller size of the bank - and therefore its database).
  - Alert/result response time comparison. **Continuous delivery of results (diefficiency metrics) does not make sense!**. With the objective to see that we can see lower response time in the case of the dp versions.

Real time simulation - problems derived:

- **Continuous delivery of results comparison does not make sense.** → In a real time simulation, for any system, results can only be emitted whenever the corresponding anomalous transaction a_i reaches the system. That happens at the same time t_i for both approaches when the input stream is simulated at real time, meaning that the result corresponding to the anomalous transaction a_i can not be emitted in any case before time t_i. Therefore, the difference in time delivery of this result between the different approaches is not expected to be high unless we make the systems to be loaded enough. **Therefore, for small sized banks this does not really make sense...**
- **Losing of alerts**: Due to scaling we are losing alerts since we have seconds precision. We will have to scale to the millisecond or nanosecond the timestamps to possibly do not loose those alerts, due to time scaling precision.
- **Although scaling, the load we are simulating is higher than real... - like for the not real time approach**

5.2.2 Approach 2: No real time simulation

IMPORTANT: WHAT DO WE WANT TO TEST?
Definition of the objectives of the experiments:

- See and compare the behavior of the system(s) with different streams (different number of cards, greater or smaller size of the bank - and therefore its database).
  Objective: see that the dp approach is better to handle bigger stream sizes.
  - Continuous delivery of results comparison (diefficiency metrics).
  - Total execution time needed to process the full stream.
  - Maximum endurance capacity of the system(s) – until which size of stream can the system work without crashing (*Hasta donde podemos llegar a aguantar con nuestro sistema. Capacidad de carga máxima.*)

Not Real time simulation - problems derived:

- **The load we are simulating is way higher than real (of course higher than for the real time approach)**
- **The reading of the input can be our bottleneck**: Try to find the fastest way to deal with it (described in 5.4).

What we do then? → Try both kinds of experiments. For the first:

- Document what I have and explain what I have seen so far.
- Continue running some more to see if I can see more differences. With more transactions and stream load.
- Try to scale to the millisecond/nanosecond timestamp precision. See if I can avoid losing alerts.

For the second: — START THEM, following the variations in the notebook (already explained)—

5.3 Experiments description

Initially, we take as reference some small Spanish banks, such as "Caja Rural de Aragón" with:

- $|ATM| \sim 200$

- $|Card| \sim 14000$

other small banks have $|ATM| \sim 200$ and around up to $|Card| \sim 10^5$. *Note that, for simplicity, we are assuming the number of bank branches as the number of ATMs and the number of clients as the number of cards.* TODO: PONER enlace a web de donde obtengo estos datos!

Regarding the size of the transaction stream, looking at some related works such as: TODO: PONER ESTAS REFERENCIAS work with a transaction stream of a size around $\sim 10^5, 10^6$.

For the transaction stream size we need to consider that our transaction generator takes as base the behavior of the clients of the Wisabi Bank Database, where each client typically produces at most $\sim$ 1 transaction per day. (TO CHECK to give the exact number).

In relation with the fraud ratio, some works like TODO: PONER ESTAS REFERENCIAS were reviewed...

5.3.1 Initial setup

Small initial graph database (gdb) size:

- $|ATM| = 50$
- $|Card| = 2000$

Transaction stream:

- `NUM_DAYS` $= 30$
- `anomalous_ratio` $= 0.02$ (2%)

This setup gives us a transaction stream of

- `total_tx` $= 39959$
- `regular_tx` $= 39508$
- `anomalous_tx` $= 451$ – note that this is actually a 1%.

Execution	Scaled	Num. cards/filter	Num. cores	Num. alerts	Time(s)
NRT	No	Baseline (all)	1	462	44.88
RT	1h	Baseline (all)	1	447	3601.65
RT	1h	500 (4 filters)	4	447	3603.25
RT	1h	200 (10 filters)	10	447	3602.71
RT	6h	Baseline (all)	1	459	21606.11
RT	6h	500 (4 filters)	4	459	21611.75
RT	12h	Baseline (all)	1	461	43211.95

Table 5: Different experimental setups results

Some nomenclature:

- NRT: Not Real Time execution
- RT: Real Time execution

Some results:

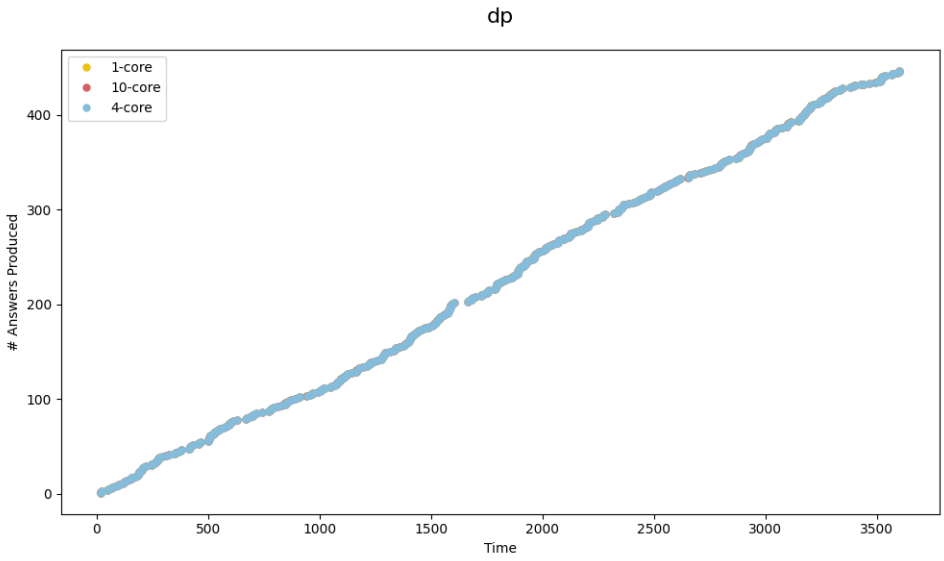


Figure 19: Trace 1h

1h scaling

Only for the first 10 results (alerts):

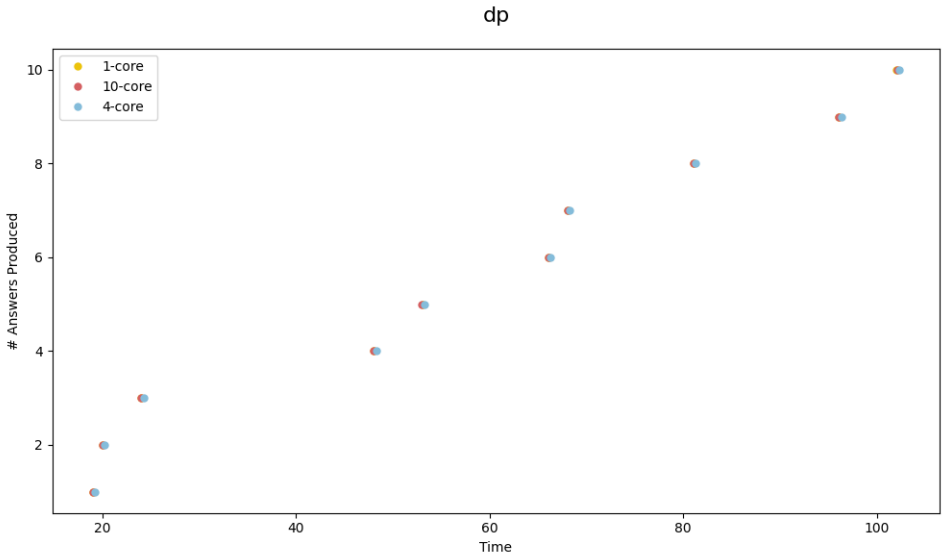


Figure 20: Trace 1h - first 10 alerts

6h scaling

We do not see any difference in the behavior between the baseline with 1 filter and 1 core approach (`RT-6h-1c-1f`) and the approach with 4 filters and 4 cores (`RT-6h-4c-4f`).

WHY? → a possible reason is that results can only be emitted whenever the corresponding anomalous transaction a_i reaches the system. That happens at the same time t_i for both approaches when the input stream is simulated at real time, meaning that the result corresponding to the anomalous transaction a_i can not be emitted in any case before time t_i. Therefore, the difference in time delivery of this result between the different approaches is not expected to be high unless we make the systems to be loaded enough.

5.3.2 Bigger instances

5.4 Input reading by chunks

- Chunk-by-Chunk: Tackling Big Data with Efficient File Reading in Chunks
- csv chunk reader - with Apache Arrow package

5.4.1 Apache Arrow

Apache arrow CSV package allows reading csv in chunks of n rows, called *records*.

The thing is that *records* / apache arrow is optimized storing the data in a columnar way (by columns). So that we can not access the original n rows

easily, but instead the columns of these rows. And therefore, from them we will need to reconstruct the rows by taking the corresponding elements from each of the columns, given the index of the corresponding row.

Good references:

- Apache Arrow and Go - Good tutorial

5.4.2 encoding/csv