TFM-FernandoMartín

Fernando Martín Canfrán

January 14, 2025

1 Introduction

TODO:

- Explicar el problema. Soluciones propuestas hasta ahora. Por qué no son suficientes. En qué consiste la nuestra.
- Definición general aplicación propuesta. Validez para PoS card fraud or internet frauds (CNP: Card Not Present frauds)... pero que hemos concretado en el caso de los ATM fraud.

  Fernando: Poner aquí contexto sobre los distintos tipos de fraude y sus estadísticas: referencia al informe SEPA ("Seventh report on card fraud typos")
- Explicar cómo está organizado el documento.

2 Related Work

TODO:

- Explicar toda la bibliografía relevante con pros y contras de cada propuesta.

3 Proposal

TODO: Formalización de nuestro sistema - planteamiento. Descripción de las distintas partes:

- Modelo de grafo
- Generación de datos
- Tipos de fraudes - Query Model
- Engine - Dynamic Pipeline
- Integración de todo el conjunto

3.1 Data Model

Fernando: Modelo de Grafo - En general. Modelo PG escogido.

Partes:

- Justificación de por qué modelamos los datos con grafo y no de otra manera...
- Graph Database Model: Property graph model. Qué es un PG, por qué se elige...
  - Simple/Easy to represent entities and their relationships with this data model.
  - Good to represent dynamic data sources... relations constantly happening: modeled as edges of a graph (AMW2024 article)
  - Good to represent evolving databases (AMW2024 article)
  - (*) Direct way to model queries related to the search of fraud patterns: as graph patterns... query model (AMW2024 article)
  - Expressiveness of queries.
  - Heuristics and indexing techniques that can be applied only if we operate in the domain of graphs: graph pattern matching and other optimizations... "translations of graphs into relations are unnatural and can not take advantage of graph-specific heuristics" (article [**PG-Graphs-at-a-time-GraphQL-** graph-at-a-time article).
- Graph Database System: Neo4j. Otras opciones (relational database), por qué se elige...
-
- Diferentes opciones
- Definir primero en general y luego definir la especificación en el caso del banco

References:

- PG: [**PG-angles2017foundations**, **PG-angles2018propertyGraphDatabaseModel**, **PG-Graphs-at-a-time-GraphQL-QueryLanguage**, **PG-exampleUsageSimeonovski**]
- Graph Databases: [**GDB-angles2008survey**, **GDB-kumar2015graph**]

Graph Database Model: Property Graph

Informally, a property graph is a directed labeled multigraph with the special characteristic that each node or edge could maintain a set (possibly empty) of property-value pairs [**angles2018propertyGraphDatabaseModel**]. In this graph, a node represents an entity, an edge a relationship between entities, and a property represents a specific feature of an entity or relationship. A more formally (as defined in [**PG-exampleUsageSimeonovski**]):

Definition 1 *A property graph $G = (V, E, \lambda, \mu)$ is a directed labeled multigraph where V is a set of nodes, $E \subseteq (V \times V)$ is a set of edges, $\lambda : V \cup E \rightarrow \Sigma$ is a function that labels nodes and edges with symbols of the alphabet Σ, and $\mu : (V \cup E) \times K \rightarrow S$ is a function that associates key-value properties, e.g., (k, s) where $k \in K$ is the key and $s \in S$ is the string value, to nodes and edges.*

Amalia: Aqui estaría bien un ejemplo pequeño, dos nodos con un arco entre ellos y varias posibles propiedades, para que quede bien claro lo que es.

Amalia: añadir algo como lo siguiente: In the Graph Database community there several popular models of property graphs. Maybe the two most famous and used ones are . . . los dos màs famosos con cita a algun sitio que diga que son los más famosos

Amalia: Muy importante: antes de explicar detalles específicos de Neo4j hay que explicar todo lo que es genérico de los property graphs y ponerlo aquí, fuera de la subsección de Neo4j

Graph Database System: Neo4j

Graph Databases: [**GDB-angles2008survey**, **GDB-kumar2015graph**]
A graph database system is a system specifically designed for managing graph-like data following the basic principles of databases systems.

Fernando: Describir el modelo de grafo escogido como data model: ”Continuously evolving data graph”

Fernando: TODO: Describir el data model - como "Continuously evolving data graph" modelado como PG en el contexto de graph database. Poner todo como conjunto. Separación y razón de la separación del modelo en stable y volatile PGs.

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.

Fernando: Poner diferenciación entre Stable and Volatile PG. Por qué separamos el modelo en 2. Mostrar el PG completo conjunto.

This property graph is a *continuously evolving data graph*, which has a 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 in data streams during a set time interval.

Fernando: Modelo de Grafo - Diseño de nuestro PG.

Property Graph Data Model Design

In what follows we describe the design of the Property Graph taken as our data model. 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.

The proposed property graph data model is represented in Figure **??**, consisting on both the stable and volatile property subgraphs merged. The details on both the stable and volatile property subgraphs are given next.

Stable Property Graph

Taking into account the reference dataset model, we designed a simplified version, as shown in Figure **??**, with the focus on representing and modeling card-ATM interactions. On it, 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.

Another option for the property graph data model representing a more common bank data model could be the one we defined in Figure **??**, which intents to capture the data that a bank system database typically gathers. It consists of a more complete and possibly closer to reality data model, although unnecessarily complex for our objectives.

As a result, our definitive 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

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

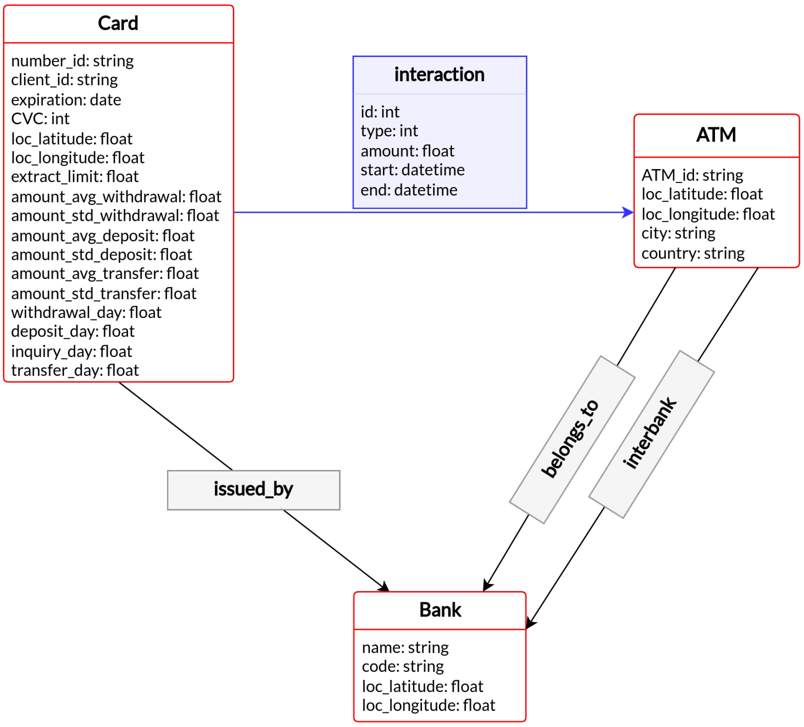


Figure 1: Complete Property Graph Data Model

headquarters, expressed in terms of *latitude* and *longitude* coordinates, as seen in Table **??**.

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

Fernando: 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 be potentially generalized to a Point Of Sale (POS) entity, allowing a more general kind of interactions apart from the current Card-ATM interaction, where also point-of-sale terminal transactions could be included apart from the ATM ones.

Fernando: Also online / card not present (CNP) transactions?

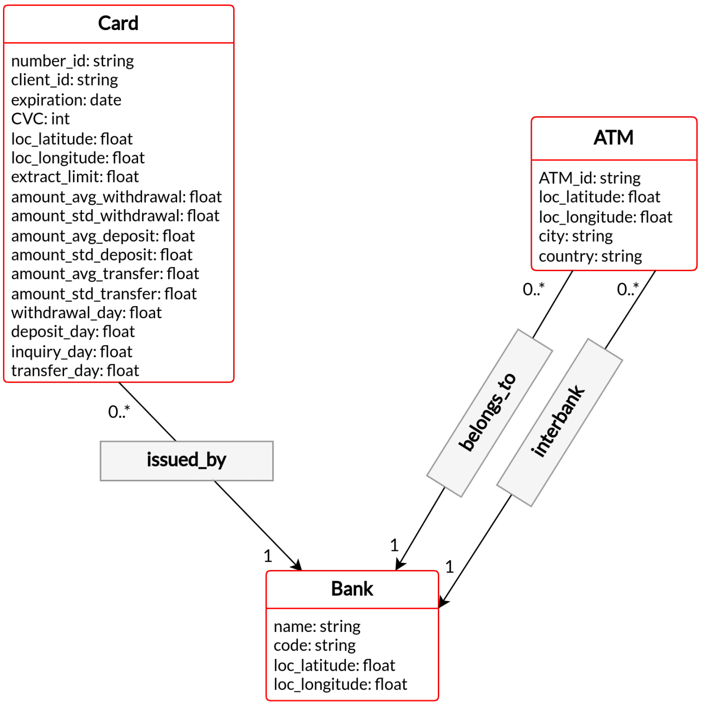


Figure 2: Stable Property Graph Data Model

Fernando: El hecho de que se podría extender a la detección de otro tipo de fraude como el de PoS (Point of Sale) comentarlo también en otro apartado más visible y/o también el las conclusiones...

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 **??**. 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 and direct description of the location of the ATMs, which will be of special interest for some of the card anomalous patterns that will be considered.

Finally, the Card node type represents the cards of the clients in the bank system. The Card node type properties, as depicted in Table **??**, consist on the card unique identifier *number_id*, the associated client unique identifier *client_id*, the card validity expiration date *expiration*, the Card Verification Code, *CVC*, the coordinates of the associated client habitual residence address *loc_latitude* and *loc_longitude* 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, related with the the amount of money a person owns. These last two properties are of special interest for some future card fraud patterns to be considered. In the first case related with interactions far from the client's habitual residence address and in the second with unusually frequent or very high expenses interactions.
Finally it contains the properties related with the *behavior* of the client, representing the usual comportment of a client in regard with its ATM usage: *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.* 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 related with anomalous client's behavior in the future.

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.

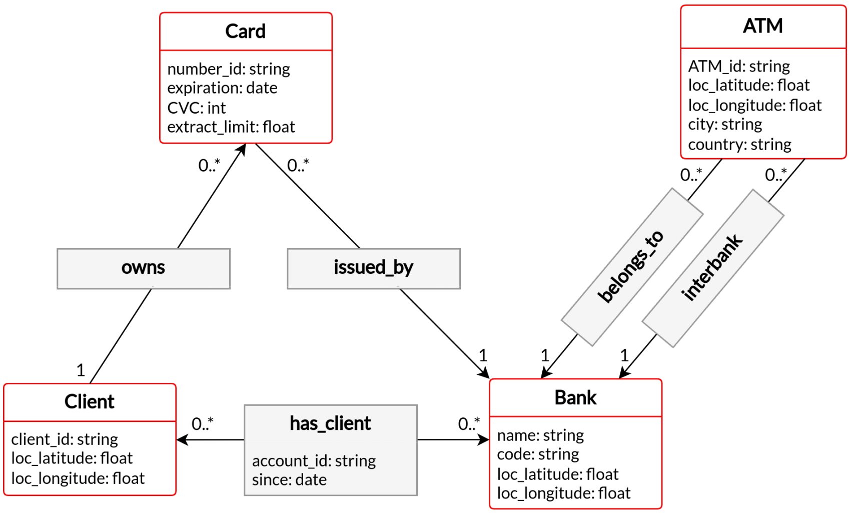


Figure 3: Alternative - Complex Stable Property Graph Data Model

Fernando: Describir el modelo más complejo? / el inicial?

The more complex alternative model, shown in Figure **??**, contains four entities: Bank, ATM, Client and Card with their respective properties, and the corresponding relationships between them. The relations are: a directed relationship from Client to Card `owns` representing that a client can own multiple credit cards and that a card is owned by a unique client, then a bidirectional relation `has_client` between Client and Bank; representing bank accounts of the clients in the bank. The relation between Card and Bank to represent that a card is `issued_by` the bank, and that the bank can have multiple cards issued. Finally, the relations `belongs_to` and `interbank` between the ATM and Bank entities, representing the two different kinds of ATMs depending on their relation with the bank; those ATMs owned and operated by the bank and those that, while not owned by the bank, are still accessible for the bank customers to perform transactions.

This model allows a more elaborated representation of what a bank system database is. As it can be seen it represents clients as an independent entity from the Card entity, and it also allows to represent bank accounts through the relation between the Client and Bank entities.

Fernando: Quitar esto ya?

Amalia: Sí, todo lo que no te sirva quítalo o comentalo ya, guarda el texto por si te puede llegar a hacer falta pero ya ve avanzando en la versión más definitiva del documento.

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 **??**) 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 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.

Volatile Property Graph

The volatile property graph consists on an abstraction of the property graph model to describe the interactions between the cards and the ATMs in the bank system (see it on Figure **??**). These interactions are going to be continuously occurring and arriving to our system as data stream.

This property graph is a subgraph of the original bank property graph model. It contains the Card and ATM entities with the minimal information needed to identify them – *number_id* and *ATM_id*, Card and ATM identifiers, respectively – between which the interaction occurs, along with additional details related to the interaction. Those identifiers are enough to be able to recover, if needed, the whole information about the specific Card or ATM entity in the stable property graph.

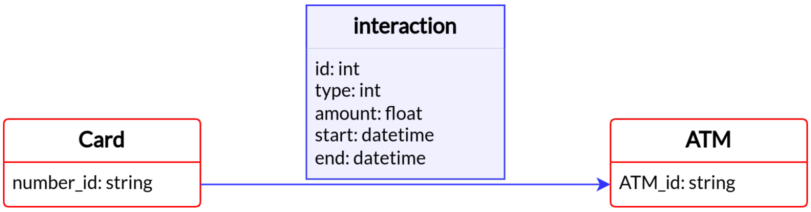


Figure 4: Volatile Property Graph Data Model

Finally, it contains 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.

Fernando: TODO: Poner tabla de propiedades de la relacion interaction como con las propiedades de las entities: Bank, Card y ATM

Fernando: TODO: Poner aqui lo de interaction start y end, la diferenciación y el por qué. En lugar de hacerlo en el apartado de la generación de transacciones

3.2 Generation of the Synthetic Dataset

As previously mentioned, given the confidential and private nature of bank data, it was not possible to find any real bank dataset. In this regard, we had to design and generate our own synthetic stable property graph bank database and stream of synthetic transactions.

Amalia: Hay que explicar porqué no servía y citar los artículos originales.

In what follows we explain all the work related with the generation of the synthetic dataset that was used to build the stable graph databases and interaction data streams for the proof of concept of our system.

Reference Dataset: Wisabi Bank Dataset

For the generation of our synthetic database and stream of transactions we considered to take as a standard data reference the *Wisabi Bank Dataset*[2], which is a fictional banking dataset publicly available in the Kaggle platform. We used it as a first base to do general customisable programs for the generation of synthetic bank datasets and streams of transactions.
The interest to use this bank dataset as a base was mainly because it is large enough and it also contained a card-ATM transactions dataset. Additionally, it provides good heterogeneity on the different kind of transactions: withdrawals, deposits, balance inquiries and transfers. The details of the *Wisabi Bank Dataset* are summarized next.

- 8819 customers.
- 50 different ATM locations.
- 2143838 card-ATM transactions records of the different customers during a full year (2022) on five different states of Nigeria (Federal Capital Territory, Lagos, Kano, Enugu and Rivers State).

The dataset consists on ten *csv* tables each with different information which is summarized on Table **??**.

Name of the Table	**Description**
enugu_transactions	Transactions of Enugu state (350251 transactions)
fct_transactions	Transactions of Federal Capital Territory state (159652 transactions)
kano_transactions	Transactions of Kano state (458764 transactions)
lagos_transactions	Transactions of Lagos state (755073 transactions)
rivers_transactions	Transactions of Rivers state (420098 transactions)
customers_lookup	Data of the different cardholders (8819 cardholders)
atm_location_lookup	Data of the different ATM locations (50 ATMs)
calendar_lookup, hour_lookup, transaction_type_lookup	Complementary data of the previous tables

Table 4: Wisabi Bank Dataset Tables Summary

The main usage that we did of this dataset was the obtention of a geographical distribution of the ATM locations and the construction of a card/client *behavior* based on the ATM-card transactions records provided.
In particular, as for the data model, we divide the creation of the synthetic dataset in two. On the one hand the creation of the stable bank dataset and on the other hand the creation of the synthetic and anomalous transactions dataset that will conform the data stream reaching our system.

[2] Wisabi bank dataset on kaggle

Stable Bank Dataset Generation

Bank dataset generator: `bankDataGenerator.py`

To do the generation of a stable bank dataset we developed the Python program `bankDataGenerator.py`. To use it we only need to enter the bank properties' values, and the number of the bank ATMs (internal and external) `n` and Cards `m` to be generated. With this program we can generate the *csv* files which define the bank dataset. A directory named `csv` will be created with the following files:

- `bank.csv`: bank entity.
- `atm.csv`: ATM entities.
- `card.csv`: card entities.
- `atm-bank-external.csv`: external ATM-bank relations.
- `atm-bank-internal.csv`: internal ATM-bank relations.
- `card-bank.csv`: card-bank relations.

To use it:

1. Ensure to have a `wisabi` named directory with the *csv* files downloaded from the *Wisabi Bank Dataset* on Kaggle.[3].
2. Ensure to have the `behavior.csv` file or run `$> python3 behavior.py` to create it.
3. Run `$> python3 bankDataGenerator.py` – Run with Python3.6 version or higher – and 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 dataset entities.

Fernando: Esto ponerlo en otr sitio:

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.

[3] Wisabi bank dataset on kaggle

Bank entity: `bank.csv`

Since a unique bank instance is considered, the values of the properties of the bank node are manually assigned, leaving them completely customisable (see an example on **??**).

```
name,code,loc_latitude,loc_longitude
Niger Bank,NIGER,6.478685,3.368442
```

Listing 1: Example of a bank.csv

ATM entity: `atm.csv`

We generate `n = n_internal + n_external` ATMs (`n_internal` ATMs owned by the bank and `n_external` external ATMs not owned by the bank, but still accessible for the bank customers to perform transactions). See an example in **??**.

- **ATM identifier**: *ATM_id*. It is assigned a different code depending on the ATM internal or external relation of the ATM with the bank:

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

- **Geographical properties**: *city*, *country*, *loc_latitude* and *loc_longitude*. They are assigned 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. 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...).

  Fernando: Put plot of the ATMs geographical distribution?

  Amalia: Si, estaria bien, un ejemplo

  With this, we assign each of the `n` ATMs *city* and *country* properties a random location/city from the *Wisabi Bank Dataset*, and we then produce random geolocation coordinates inside the bounding box of the city location to set as the *loc_latitude* and *loc_longitude* properties of the ATM.

```
ATM_id,loc_latitude,loc_longitude,city,country
NIGER-0,12.124651,8.543515,Kano,Nigeria
NIGER-1,12.148756,8.481764,Kano,Nigeria
EXT-0,8.941474,7.526291,Abuja,Nigeria
EXT-1,4.816251,7.010188,Port Harcourt,Nigeria
```

Listing 2: Example of atm.csv

Card entity: `card.csv`

We generate a total of `m` cards that the bank manages. For each of them the assignment of the different properties is done as explained next. An example of a Card data item can be seen in **??**.

- **Card and client identifiers**:

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

- ***expiration* and *CVC* properties**: they are not relevant, could be empty value properties indeed or a same toy value for all the cards. 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 usual 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* properties**: these are properties of special interest both when performing the generation of the synthetic transactions of each of the cards and also for the detection of future possible fraud patterns. The defined *behavior* properties are shown in Table **??**. They refer about metrics related with four different types of operations: withdrawal, deposit, balance inquiry and transaction. These operations will be the ones that, as in the base dataset, we will also consider that a customer can perform when we generate our synthetic transaction dataset.

Behavior Property	**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 5: *Behavior* properties

The behavior properties' values are assigned to each of the cards by taking a random behavior *row* from the `behavior.csv` file. This `behavior.csv` contains the gathered behavior metrics of each of the customers of the *Wisabi Bank Dataset*. It was generated by using the Python program `behavior.py`. This program creates the behavior of each of the original customers of the *Wisabi Bank Dataset* by doing a summary of all its transaction records on this base dataset.

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.

- **Card money amount extraction limit property**: *extract_limit*. We set it up by setting an upper bound based on the *amount_avg_withdrawal* behavior metric of the card. Other possible ways could be chosen for assigning a value to this property.

$$\mathit{extract\_limit} : \mathit{amount\_avg\_withdrawal} * 5$$

```
number_id,client_id,expiration,CVC,loc_latitude,loc_longitude,extract_limit
c-NIGER-0,0,2050-01-17,999,8.92926,7.398833,121590.9
```

Listing 3: Example of card.csv (Part 1)

```
amount_avg_withdrawal,amount_std_withdrawal,withdrawal_day
24318.18,28174.96,0.2411
```

Listing 4: Example of card.csv (Part 2)

```
amount_avg_deposit,amount_std_deposit,deposit_day,inquiry_day
11500.0,5889.33,0.0548,0.0493
```

Listing 5: Example of card.csv (Part 3)

```
amount_avg_transfer,amount_std_transfer,transfer_day
21448.28,20500.15,0.0795
```

Listing 6: Example of card.csv (Part 4)

ATM-Bank `interbank` relation: `atm-bank-external.csv`

To represent the `interbank` relations between the bank and the external ATMs. Taking the unique ids of each of the related entities, the bank *code* and the *ATM_id*. Example on ??.

```
code,ATM_id
NIGER,EXT-0
NIGER,EXT-1
```

Listing 7: Example of a atm-bank-external.csv

ATM-Bank `belongs_to` relation:`atm-bank-internal.csv`

To represent `belongs_to` relations between the bank and the internal ATMs. Taking the unique ids of each of the related entities, the bank *code* and the *ATM_id*. Example on ??.

```
code,ATM_id
NIGER,NIGER-0
NIGER,NIGER-1
NIGER,NIGER-2
```

Listing 8: Example of a atm-bank-internal.csv

Card-Bank `issued_by` relation: `card-bank.csv`

To represent the `issued_by` relations between the bank and the card entities. Taking the unique ids of each of the related entities, the bank *code* and the Card id: *number_id*. Example on ??.

```
code,number_id
NIGER,c-NIGER-0
NIGER,c-NIGER-1
NIGER,c-NIGER-2
```

Listing 9: Example of a card-bank.csv

Transactions Set Generation

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 with the form of an *interaction* edge/relation of the volatile subgraph (see **??**) matching one Card with one ATM of the bank database. In what follows we explain the generation of the transaction stream that we employed in order to test our system. Note that, in any case, this is only a possible proposal done with the intention of reproduce a close-to-reality bank transaction stream.

Fernando: TODO: Explain why we did this division open/close of the interaction...Dónde poner esta explicación?. Es importante, pues es una de las claves que pueden diferenciar/hacer a nuestro sistema diferente, capaz de hacer una detección de fraude en tiempo real

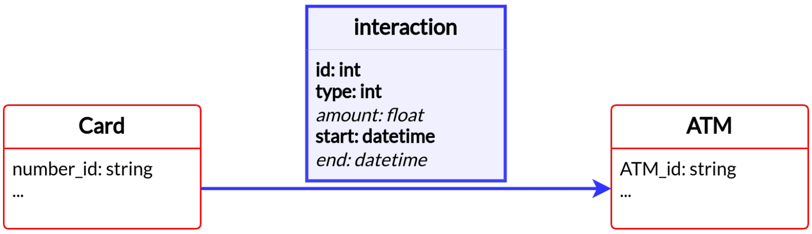


Figure 5: *Opening* interaction edge

Fernando: Esto ponerlo en la definicion de interaction dentro de la definicion de volatile property graph del punto anterior

It is important to remark that, as in our definition of the input data stream of the DP_{CQE}, we generate two edges per transaction/`interaction` relation – the *opening* and the *closing* edges – which both will constitute a single `interaction` relation. The *opening* edge (Figure **??**) will be the indicator of the beginning of a new interaction between the matched Card and ATM, it contains the values of the properties related with the starting time *start*, the interaction *type* as well as the `id`. The *closing* edge (Figure **??**) will indicate the end of the interaction, completing the values of the rest of the properties of the interaction: *end* and *amount*.
With this division of the interaction relation in two edges we are simulating that on each transaction, our system receives an initial message when the interaction starts and a final message once the interaction is finished on the ATM. Allowing us to develop a system that is able not only to detect anomalous scenarios on interactions that have already been produced/closed, but also to act in real time before the anomalous interaction detected is actually finished.

We divide the generation of the transaction set in the generation of two subsets: the regular transaction set and the anomalous transaction set. The regular transaction set consists of the *ordinary/correct* transactions, that are guaranteed to not produce any anomalous scenarios, whereas the anomalous transaction set is composed of the

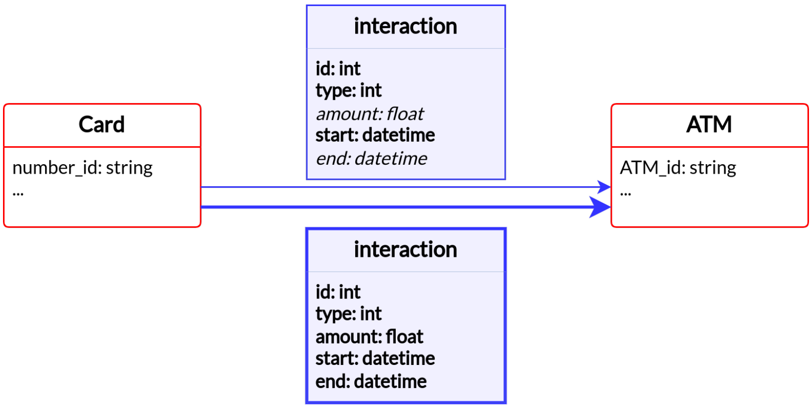


Figure 6: *Closing* interaction edge

irregular/anomalous transactions that are intentionally created to produce anomalous scenarios. The main reason to do this separation on the generation is to divide the creation of the full transaction stream in two steps. First the creation of the stream of the regular transaction set, having the control to ensure that no anomalous fraud scenarios are produced in between the transactions of this set. And second, and only after the creation of the regular transaction set, we create the anomalous transaction set, creating transactions that originate anomalous fraud scenarios over the regular transaction set.

Regular Transaction Set

The main idea of the creation of this set, is to produce a set of ordinary transactions for each of the cards that do not produce any anomalous scenarios between them. Depending on the fraud types considered some constraints need to be added when doing the generation of this transaction stream in order to avoid the accidental creation of these kind of fraud patterns on this set. After this set is created, the transactions producing anomalous scenarios related with each specific fraud pattern will be produced and injected to compose the final stream of transactions.
So far, for the fraud patterns that we are considering the constraints that we need to impose on the generation of the regular transaction set are:

I. Fraud pattern I: No two consecutive transactions in different ATM locations can be produced with an insufficient feasible time difference.

The transaction stream is generated for a customisable `NUM_DAYS` number of days starting in a `START_DATE` for each of the cards on our bank network. For each card we take its behavior (see **??**) to determine the number and type of interactions performed in the defined days time interval: $[\texttt{START\_DATE}, \texttt{START\_DATE} + \texttt{NUM\_DAYS}]$. The interactions are generated by linking the card to ATMs that are no farther than `MAX_DISTANCE_SUBSET_THRESHOLD` kilometers from the residence location of the client of the card `residence_loc`, given by the location coordinates of the card entity: $(loc\_latitude, loc\_longitude)$. Nevertheless, in a simpler version of the transaction generator program we also consider avoiding this limitation and allow to link the card to any ATM of the bank dataset. Finally, the interactions are distributed along the defined time interval [$\texttt{START\_DATE}, \texttt{START\_DATE} + \texttt{NUM\_DAYS}$] respecting the constraint (I) related with the Fraud Pattern I: limit the time distance between two consecutive transactions so that, for a card, there are no two consecutive transactions in different ATM locations with an insufficient feasible time difference.

As a summary of the procedure a pseudocode of the transaction generator is depicted in Algorithm **??**, of which some of its parts are later explained.

Fernando: TODO: Cambiar, no me gusta lo del residence_loc, no se define así la property en la card!

Algorithm 1 Regular Transactions Generation

```
$\texttt{id} \leftarrow 0$
for card in cards do
    $\texttt{ATM\_subset}, \overline{\texttt{ATM\_subset}} \leftarrow \text{createATMsubset}(\texttt{residence\_loc})$
    $\texttt{t\_min\_subset} \leftarrow \text{calculate\_t\_min\_subset}(\texttt{ATM\_subset})$
    $\texttt{num\_tx} \leftarrow \text{decide\_num\_tx}()$
    $T \leftarrow \text{distribute}(\texttt{num\_tx}, \texttt{t\_min\_subset})$
    for $t_i$ in $T$ do
        $\texttt{ATM}_i \sim \texttt{ATM\_subset}$
        $\texttt{start}_i \leftarrow t_i.start$
        $\texttt{end}_i \leftarrow t_i.end$
        $\texttt{type}_i \leftarrow \text{getType}()$
        $\texttt{amount}_i \leftarrow \text{getAmount}(\texttt{type}_i)$
        $\texttt{id}_i \leftarrow \texttt{id};\ \texttt{id} \leftarrow \texttt{id} + 1$
        $\text{createTransaction}(\texttt{id}_i, \texttt{ATM}_i, \texttt{start}_i, \texttt{end}_i, \texttt{type}_i, \texttt{amount}_i)$
    end for
    $\text{introduceAnomalous}(\texttt{ATM\_subset}, \overline{\texttt{ATM\_subset}})$ {Anomalous transaction set}
end for
```

1. **Creation of the ATM subset**

   $\texttt{ATM\_subset}, \overline{\texttt{ATM\_subset}} \leftarrow \text{createATMsubset}(\texttt{residence\_loc})$

   `ATM_subset` is the subset of ATMs of the stable bank dataset, in which we will allow the interactions of each of the cards to occur. We set a limit on the size of this subset, considering only a maximum ratio of the total number of ATMs on the dataset ($\texttt{MAX\_SIZE\_ATM\_SUBSET\_RATIO} \in [0, 1]$), so that only a certain amount of the ATMs are included on it:

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

   There are two options for the construction of this subset:

   - **Distance to cardholder residence location selection**: Only the closest $|\texttt{ATM\_subset}|$ ATMs to the cardholder residence location `residence_loc` = *(loc latitude, loc longitude)* are included in the subset. These ATMs are considered to be *usual* for the cardholder, in terms of its location distance. Apart from the subset size limitation, a maximum distance constraint defined by the `MAX_DISTANCE_SUBSET_THRESHOLD` parameter can be imposed:

$$\texttt{ATM\_subset} = \{\texttt{ATM} \mid \texttt{distance(ATM, residence\_loc)} \\ \leq \texttt{MAX\_DISTANCE\_SUBSET\_THRESHOLD}\}$$

Fernando: TODO: CAMBIAR ESTA IMAGEN

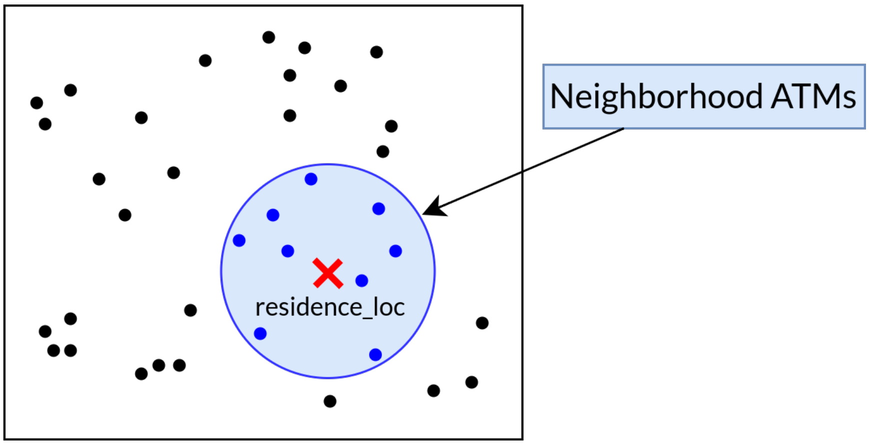


Figure 7: `Neighborhood` ATM subset

- **Random selection**: The `ATM_subset` is built by randomly selecting |`ATM_subset`| ATMs from the stable bank dataset.

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

`t_min_subset` is the minimum threshold time needed to respect between any two consecutive transactions of a card in the regular transaction set.

That is, `t_min_subset` is the minimum time distance between the end of a transaction and the start of the next consecutive transaction of a card, needed to guarantee in order to ensure that the constraint (I) is respected on the generation of the regular transactions set.

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

To calculate 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 locations can be traveled in the case of regular transaction scenarios: `REGULAR_SPEED`.

3. **Decide the number of transactions to be generated `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 defined days time interval [`START_DATE`, `START_DATE`+ `NUM_DAYS`]:

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

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

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

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

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

Along the selected time interval $[\texttt{START\_DATE}, \texttt{START\_DATE} + \texttt{NUM\_DAYS}]$ 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, respecting the constraint (I) in order to guarantee that, no two consecutive transactions tx_i and tx_{i+1} performed in any of the ATMs of the `ATM_subset` are at a time distance lower than `t_min_subset`. Specifically, the transaction times are generated guaranteeing:

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

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:

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

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

Fernando: TODO: Explicar la variante (del generador simplificado `txGenerator-simplified.py`) en el que este distribute_tx se cambia de tal forma que: with random order (like original version) if num holes/2 ¿ needed holes, otherwise we introduce the transactions ordered in the time interval increasingly ordered one after the other

5. **Decision on the specific values of each transaction properties:** Once all the previous steps are done, the specific values for the properties of each of the `num_tx` transactions can be decided (this corresponds to the lines 7-15 in the algorithm pseudocode **??**). Therefore for each of the transactions:

   - **Link to a random ATM of the `ATM_subset`**
     $\texttt{ATM}_i \sim \texttt{ATM\_subset}$
   - **Obtain its corresponding *start* and *end* time property values from the T time distribution**
     $\texttt{start}_i \leftarrow t_i.start, \quad \texttt{end}_i \leftarrow t_i.end$
   - **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:

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

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

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

   - **Assign a transaction *amount***
     $\texttt{amount}_i \leftarrow \text{getAmount}(\texttt{type}_i)$

     The transaction *amount* is assigned depending on the *type* of the current transaction, based on the card behavior properties:

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

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

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

An example of a transaction data item can be seen in **??**, where both the interaction opening and interaction close of the *transaction_id* 2804 can be observed.

```
transaction_id,number_id,ATM_id,transaction_type,transaction_start,
transaction_end, transaction_amount
2804,c-NIGER-148,NIGER-40,0,2018-04-01 00:00:47,,
2804,c-NIGER-148,NIGER-40,0,2018-04-01 00:00:47,2018-04-01 00:04:43,26886.73
```

Listing 10: Example of transaction-all.csv

Fernando: TODO: SIGO AQUÍ!

Fernando: TODO: Explain the simplified version of the txGenerator...:

For the medium size experiments (500000 cards), to generate the stream of tx, we needed to simplify this process in order to be able to generate a stream in a feasible amount of time. In particular we used the simplifed version of the `txGenerator.py`: `txGenerator-simplified.py` → with a random ATM-subset instead of a closest to client ATM-subset. Also variation on the transaction distribution times, with random order (like original version) if $num\_holes/2 > needed\_holes$, otherwise we introduce the transactions ordered in the time interval increasingly ordered one after the other.

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.

- 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

- Anomalous generator:
  - NO-Overlapping assumption - Explain
  - Any type of tx to produce the fraud -¿ does not matter the type for the FP1.

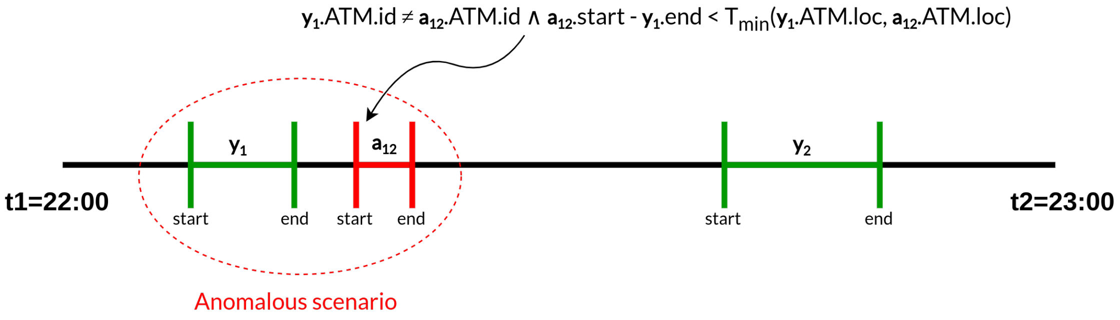
Anomalous Transaction Set

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 **??**, 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...).



y1.ATM.id ≠ a12.ATM.id ∧ a12.start - y1.end < Tmin(y1.ATM.loc, a12.ATM.loc)
y1
a12
y2
t1=22:00
t2=23:00
start
end
start
end
start
end
Anomalous scenario

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**: Apart 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 **??**.

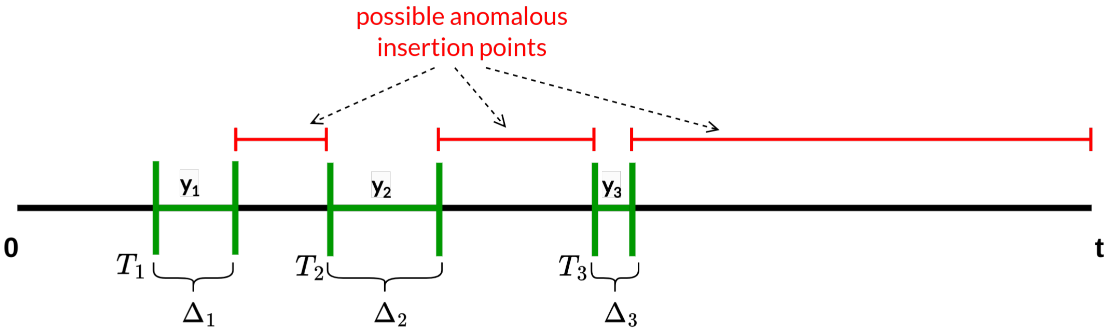


Figure 9: Considered possible injection points of anomalous transactions of fraud type I

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 **??** 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($\texttt{ATM\_subset}, \overline{\texttt{ATM\_subset}}$)
$\texttt{num\_anomalous} \leftarrow \texttt{num\_tx} * \texttt{ANOMALOUS\_RATIO\_1}$
$i \leftarrow 0$
while $i < \texttt{num\_anomalous}$ do
    $\texttt{ATM}_i \sim \overline{\texttt{ATM\_subset}}$
    $prev_i, next_i \leftarrow$ randomUniquePosition($\texttt{num\_tx}$)
    $t_i \leftarrow$ getTime($prev_i, next_i$)
    $\texttt{start}_i \leftarrow t_i.start$
    $\texttt{end}_i \leftarrow t_i.end$
    $\texttt{type}_i \leftarrow \texttt{getRandomType()}$
    $\texttt{amount}_i \leftarrow \texttt{getAmount()}$
    $\texttt{id}_i \leftarrow \texttt{id}$; $\texttt{id} \leftarrow \texttt{id} + 1$
    $\texttt{createTransaction}(\texttt{id}_i, \texttt{ATM}_i, \texttt{start}_i, \texttt{end}_i, \texttt{type}_i, \texttt{amount}_i)$
    $i \leftarrow i + 1$
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 **??**, 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**

Fernando: Poner otras opciones consideradas? - ver texto comentado justo debajo

Fernando: TODO: Explain the different program versions done and how to use each. Files generated. Show example of the csv.

Transaction dataset generator: `txGenerator.py`

To do the generation of a transaction stream dataset we developed a Python program `txGenerator.py`. The program contains some parameters used for the customization of the generated transaction stream.

Fernando: TODO: List the parameters to configure/set up the generation of the tx stream

To use it:

1. Ensure to have a `csv` named directory with the *csv* stable bank dataset files on which we want to simulate a transaction stream (use the bank data generator to produce it).

2. Run `$> python3 txGenerator.py <outputFileName>` – Run with Python3.6 version or higher – introducing *outputFileName* as an argument to name the transaction dataset files to be generated.

The program generates a `tx` directory with the *csv* files representing the transaction stream dataset:

- `<outputFileName>-all.csv`: joint regular and anomalous dataset.

- `<outputFileName>-regular.csv`: regular transaction dataset.

- `<outputFileName>-anomalous.csv`: anomalous transaction dataset.

Fernando: TODO: Explicar la versión simplificada. Random ATM subset. Actualización de la función de dsitribute_tx, = if $num\_holes/2 \geq needed\_holes$...

Transaction dataset generator: `txGenerator-simplified.py`

3.3 Population of the Graph Database

3.3.1 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`.

3.3.2 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 11: 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 ??. 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.

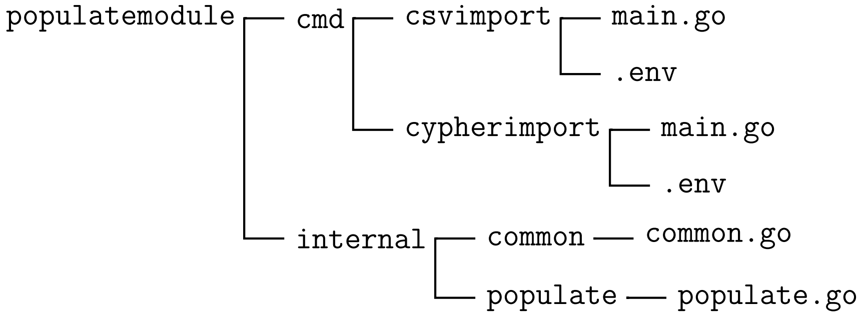


Figure 10: `populatemodule` file structure

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 12: 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 13: 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 14: 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 15: 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 16: 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 17: 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...

3.4 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.

3.5 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 ;)

3.6 The Query Model

- General formalization: Continuous queries.
- Fraud patterns identified - Definition and their formalizations
- Profundizar más en el que se ha probado. Describir implementación.

Fernando: Recabar enlaces de definiciones de fraudes bancarios pasados por whats

The Query Model: Continuous Query Model

Fernando: Formalización del query model. Completar más.

In the context of our application, taking into account that the input data of our system takes the form of a continuous data stream, we categorize our query model under the *continuous query* model [**CQ-babu2001continuous**, **CQ-zaniolo2012logical**]. The continuous query model is the ideal query model for applications considering queries evaluated over data streams (unbounded sequences of timestamped data items), in contrast with classical query evaluation processes, where the data to query is stable, with small or infrequent updates.
Although, part of the data source of our application is stable (the stable bank database), the input of our system consists of a data stream, data in motion and continuously changing, as it is the ATM transactions input data stream.
In our work, we tackled the problem of evaluating continuous queries corresponding to anomalous patterns of ATM transactions against a continuously evolving PG representing a bank database. The bank database is continuously evolving due to the input ATM transactions data stream that is continuously receiving. The anomalous patterns of ATM transactions are identified in the volatile (PG) subgraph of the considered database. With this, a query on our PG database can be defined as a PG graph pattern with constraints over some of its properties. Evaluating such a query consists on identify if there is a subgraph of the database that matches the given graph pattern and satisfies its constraints.

The Query Model: Fraud Patterns Definition

It is not trivial to establish what is and in which circumstances an ATM transaction can be considered anomalous. Based on a work that have addressed this characterization [**FP-magdalena2021artificial**] we intend to find a proper characterization and then define the graph patterns associated to these anomalies. The exact topology of an anomaly will depend on its own nature. Moreover, definition of patterns can be beyond ATM transactions by considering online card transactions. In what follows, we propose a characterization of some possible anomalous patterns of ATM transactions and the definition of their associated PG graph patterns.

I. Card cloning characterization

II. Lost-and-stolen card characterization

III. Other possible fraud scenarios

I - Card Cloning Characterization

Definition references:

- [**FP-unit21˙card˙cloning**]
-

Card cloning can be defined as a "type of fraud in which information on a card used for a transaction is covertly and illegally duplicated. Basically, it's a process thieves use to copy the information on a transaction card without stealing the physical card itself. This information is then copied onto a new or reformatted card, allowing criminals to use it to make fraudulent purchases or gain unauthorized access to a person's accounts."'[**FP-unit21˙card˙cloning**].
There are many possible ways to detect a card cloning scenario, among others the analysis of the customer's transaction data to construct typical transaction behaviors so to be able to detect uncommon transaction behaviors. However, in our work we propose an alternative possible method based on a PG graph pattern detection.
In particular, the method consists on detecting card-ATM activity of the same card at different ATMs taking place within an unfeasible time distance difference. That is, when at a certain time a transaction is made in an ATM and after that, another transaction is started with that same card in a different ATM location such that the distance between the two ATM locations is impossible to cover in the time difference between the two transactions. The detection of this anomalous scenario is represented on the PG graph pattern of the Figure **??**.

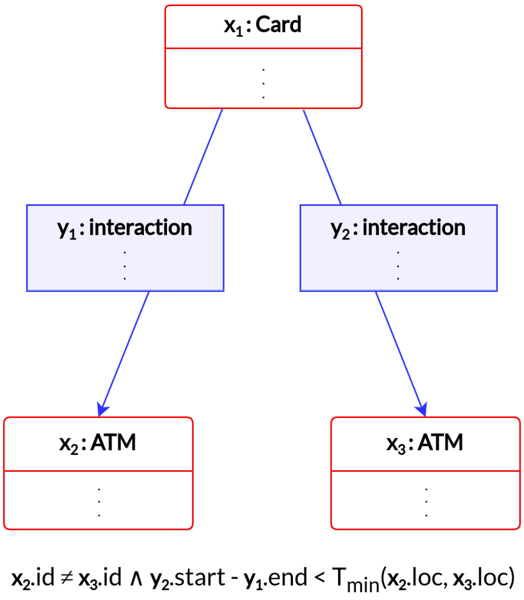


Figure 11: Card Cloning Characterization - Graph Pattern

The pattern consists on Card entity x_1, having two interaction relations y_1 and y_2 with two different ATMs x_2 and x_3, respectively, such that the time difference between the ending time of the first interaction $y_1.end$ and the starting time of the second interaction $y_2.start$, is not sufficient to cover the minimum time needed to travel from the first to the second ATM location $T_{min}(x_2.location, x_3.location)$. As a whole:

$$x2.id \neq x3.id \ \wedge \ y_2.start - y_1.end < T_{min}(x_2.location, x_3.location)$$

where $x_2.location$ location represents the location coordinates pair of the x_2 ATM: $x_2.location = (x_2.loc\_latitude, x_2.loc\_longitude)$. Same for the x_3 ATM.

Fernando: TODO: Poner qué es / cómo se calcula el tmin (la formula, que es parametrizable en base a la velocidad max decidida...

An example of this kind of anomalous card-ATM interaction, could be one as represented on Figure **??**, in which an ATM interaction with a certain card is finished at time 22:14 in Barcelona, and then another interaction with that same card starts at time 22:56 of that same day in Madrid. Clearly this should be reported as this kind of anomalous scenario since it is impossible, for the time being, to cover the distance between these two cities in that time difference.

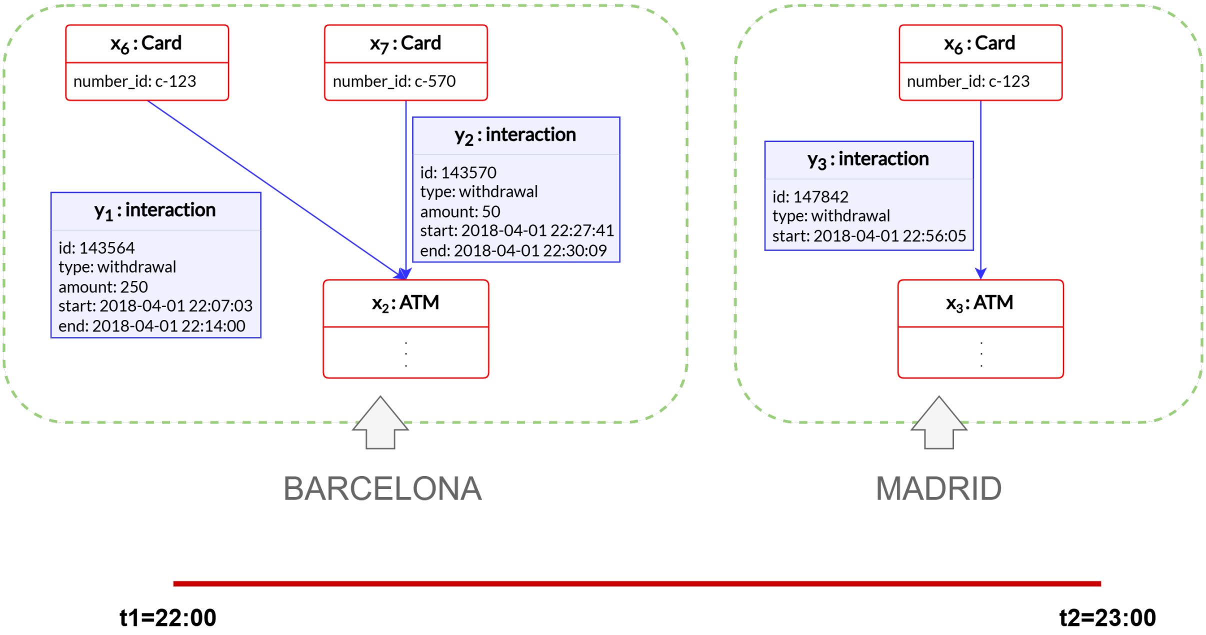


Figure 12: Card Cloning Characterization - Example

II - Lost-and-Stolen Card Characterization

Fernando: Referencias para este tipo de anomalía. Definir mejor... Completar

Definition references:

- https://www.datavisor.com/wiki/lost-or-stolen-card-fraud/
- https://www.americanexpress.com.kw/en-kw/fraud-protection-center/lost-and-stolen-card-fraud/

”Lost-and-stolen card is the fraud scenario produced when a card is physically stolen or is lost, and is then used by a criminal, posing as you, to obtain goods and services” [**FP-lost-and-stolen-americanexpress2025**].
A possible way that we propose to detect this kind of scenario is through the tracking of a typical behavior that it is produced when the card is used by the criminal. That is, when obtained, the fraudster tries to do as many as possible money withdrawals in different ATMs before the owner of the card realises and asks the bank to freeze the card. The detection of this kind of fraud scenario is modeled with a PG graph pattern as the one represented on Figure **??**.

Fernando: DUDA: Indicar en el dibujo que el tipo de interacción/operación es el de withdrawal?

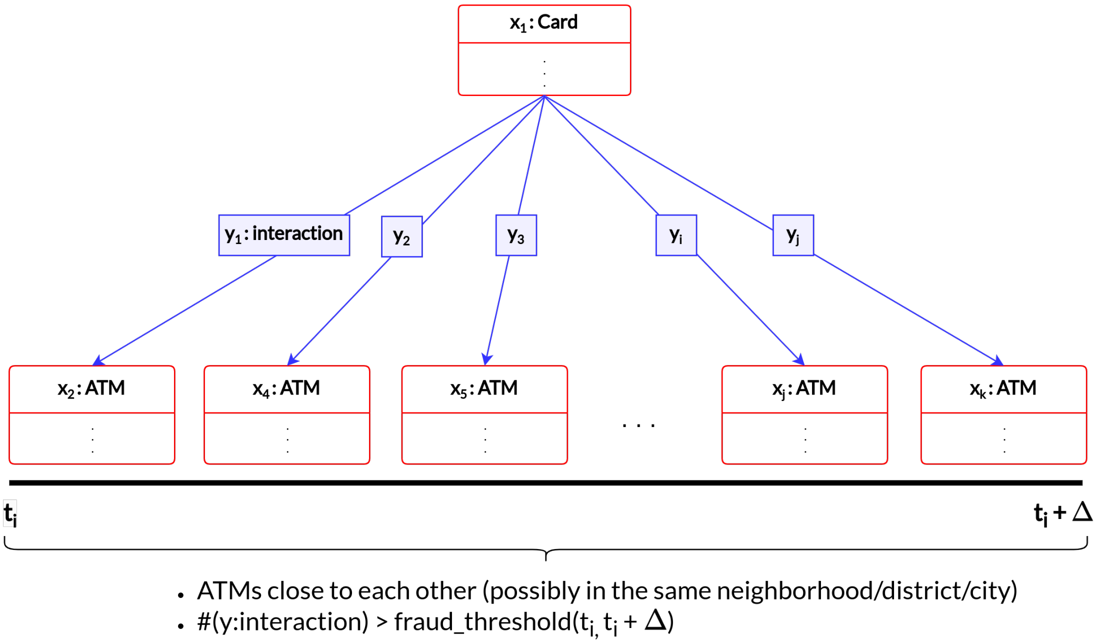

x₁ : Card
y₁ : interaction
y₂
y₃
yᵢ
yⱼ
x₂ : ATM
x₄ : ATM
x₅ : ATM
. . .
xⱼ : ATM
xₖ : ATM
tᵢ
tᵢ + Δ
• ATMs close to each other (possibly in the same neighborhood/district/city)
• #(y:interaction) > fraud_threshold(tᵢ, tᵢ + Δ)

Figure 13: Lost-and-Stolen Card Characterization - Graph Pattern

On it we define a Card entity x_1 having a number k of interactions y at different ATMs $x_2 ... x_k$ within a time interval $[t_i, t_i + \Delta]$, where $t_i = y_1.start$ and $t_i + \Delta = y_k.start$, such that k is considered to be an usual high number of withdrawals for that time interval. A reference for the usual number of withdrawals on a certain time interval for a specific cardholder can be obtained from the gathered cardholder behavior *withdrawal_day* card entity property. Another indicator of this scenario to be considered could also be the *amount* value of the withdrawal operations performed, which is normally a low value to prevent that the card owner realises.

Fernando: Otro tipo de fraude que comentamos es el de detectar más de una extracción en un tiempo dado... realmente creo que quedaría dentro de este mismo tipo de fraude.

III - Other possible fraud scenarios

Some other anomalous scenarios for which more graph patterns could be defined are:

- **Anomalous location usage**: When a transaction is made in a location out of the threshold distance of the usual/registered address of the cardholder.
- **Anomalous number of operations**: Related with the II pattern characterized, we could also define a graph pattern related with a higher than average number of operations of any kind (withdrawal, inquiry, transfer or deposit) for a cardholder in a certain time interval.
- **Anomalous high expenses:** Similar to the II pattern, but in this case, not considering only the number of the withdrawal operations performed on a certain time interval, but the amount of the withdrawal operations on a certain time interval. This could indicate an anomalous behavior of the cardholder, withdrawing an amount of money way higher for a considered time interval.

The Query Model: Fraud Patterns Algorithmic Description

So far, among all the characterized anomalous scenarios as PG graph patterns, for our proof of concept, we implemented the detection of the first defined graph pattern, the graph pattern related with the card cloning characterization.

Fernando: En lo que sigue se explica:
1. el algoritmo de inicio en el que se tenía un subgrafo de todas las interacciones que llegaban por cada una de las cards. Se hacía el check de una interacción entrante con cada una de las interacciones del subgrafo.
2. Como se vio que se podía simplificar (bastaba con hacer el check con la más reciente del subgrafo, se simplificó el subgrafo (almacenando solo la interaccion más reciente) y también el algoritmo...
DUDA: Pongo directamente el modelo/forma final o dejo todo?

Amalia: Yo pondría el final pero si te gusta más que esté todo, déjalo todo.

Fernando: Esta parte creo que sería más conveniente explicarla en el apartado de Implementation del $\mathsf{DP}_{\mathsf{ATM}}$ porque ahi ya se han explicado los subgrafos, los filter stages... y otros conceptos necesarios... REORDENAR

I - Card Cloning Graph Pattern Algorithm

Fernando: Faltaría explicar que se tiene un subgrafo por cada card donde se van acumulando/almacenando todas las relaciones/interacciones de la card...

A first algorithmic proposal to detect this kind of fraud pattern is the one shown in the algorithm **??**. 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 any fraud checking operation 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
fraudIndicator $\leftarrow$ False
$i \leftarrow |S|$
while $i > 0$ and fraudIndicator $=$ False do
    $e_i \leftarrow S[i]$
    t_min $\leftarrow$ obtain_t_min($e_i, e_{new}$)
    t_diff $\leftarrow e_{new}.start - e_i.end$
    if t_diff $<$ t_min then
        createAlert($e_i, e_{new}$)
        fraudIndicator $\leftarrow$ True
    end if
    $i \leftarrow i - 1$
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 **??**. 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
$last \leftarrow |S|$
$e_{last} \leftarrow S[last]$
t_min $\leftarrow$ obtain_t_min($e_{last}, e_{new}$)
t_diff $\leftarrow e_{new}.start - e_{last}.end$
if t_diff < t_min then
    createAlert($e_{last}, e_{new}$)
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 **??**, 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.

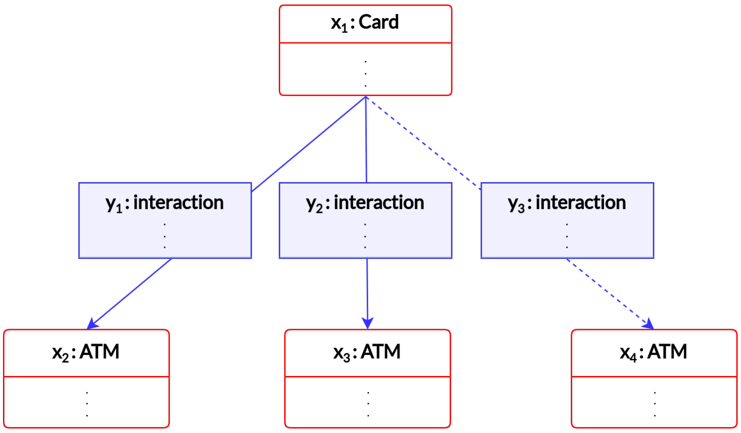


Figure 14: Subgraph S of a card – Fraud Pattern I

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:

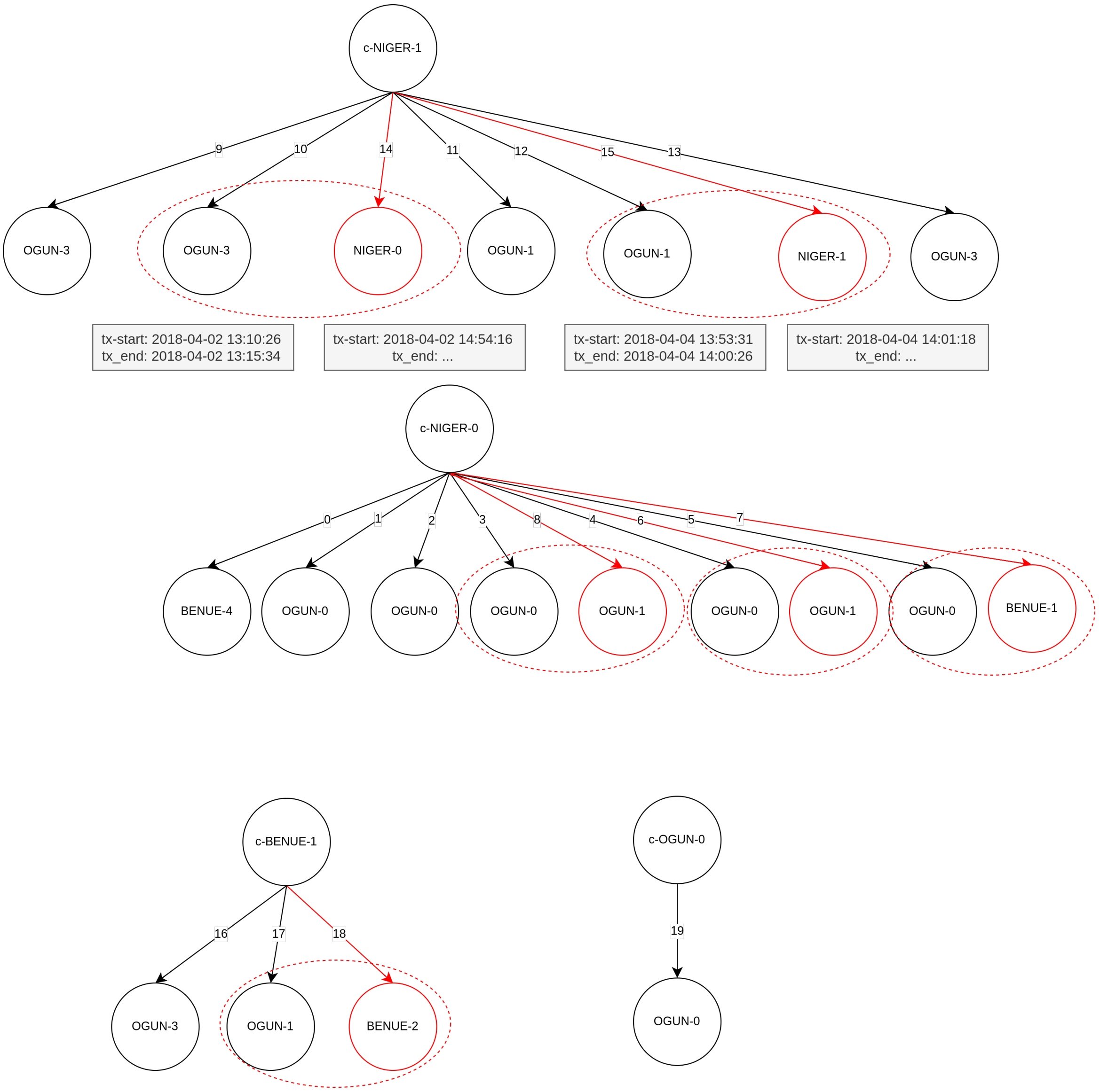
$$\#TX\_ANOM \leq SCENARIOS \leq 2 * \#TX\_ANOM$$

  TODO: Poner dibujo y explicar mejor

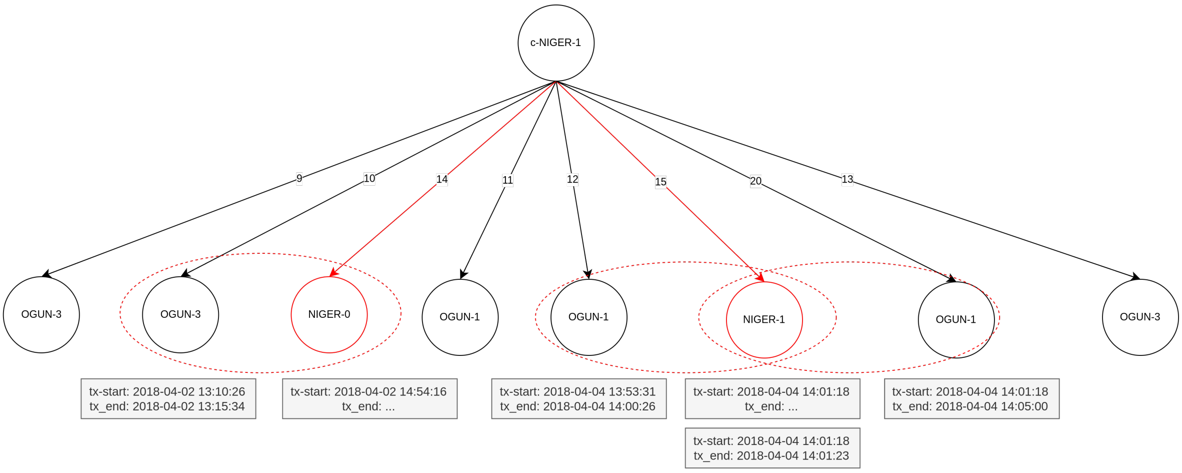
3.7 Initial Tests

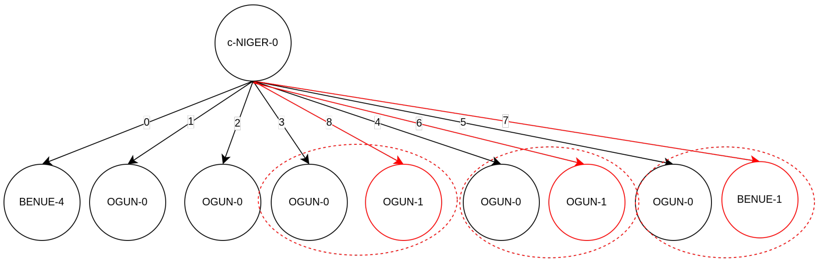
In the following we describe the preliminar tests that were done to check the correct expected behavior of the FP_1 algorithm.

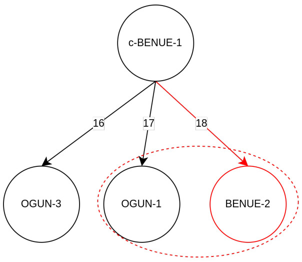
3.7.1 FP_1 - Test 1

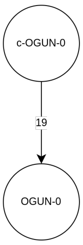


3.7.2 FP_1 - Test 1.1


c-NIGER-1
9
10
14
11
12
15
20
13
OGUN-3
OGUN-3
NIGER-0
OGUN-1
OGUN-1
NIGER-1
OGUN-1
OGUN-3
tx-start: 2018-04-02 13:10:26
tx_end: 2018-04-02 13:15:34
tx-start: 2018-04-02 14:54:16
tx_end: ...
tx-start: 2018-04-04 13:53:31
tx_end: 2018-04-04 14:00:26
tx-start: 2018-04-04 14:01:18
tx_end: ...
tx-start: 2018-04-04 14:01:18
tx_end: 2018-04-04 14:05:00
tx-start: 2018-04-04 14:01:18
tx_end: 2018-04-04 14:01:23







3.8 Dynamic Pipeline Paradigm

To explain:

- PP (Pipeline Parallelism) computational model. The definition
- DP stages y un poco que hace cada stage
- Problemas que ya se han resuelto con este paradigma. Referenciar.

Fernando: Definición - Tomada de TFM Dani y J Pablo... completar mas?

Amalia: Yo creo que puedes citar ambas tesis y también el paper de Edelmira y explicar lo que es pero no hace falta que entres en demasiado detalle general, eso sí, los detalles de cómo lo usas para lo tuyo sí deben estar completos.

In the context of Stream Processing, many data driven frameworks have emerged to address the management of continuous data streams. Dynamic data processing, characterized by the adaptive and responsive manipulation of large datasets in real time, and incremental generation of results, represent pivotal approaches.

One such model for Stream Processing is the Dynamic Pipeline Paradigm (DPP) [**DP-pasarella2024computational**]. The DPP is a PP (Pipeline Parallelism) data driven computational model that operates as a one-dimensional, unidirectional chain of stages connected by means of data channels. Essentially, the paradigm establishes a computational model rooted in the deployment of a linear pipeline consisting of a chain of stages structure called Dynamic Pipeline (DP). It stretches and shrinks depending on the spawning and the lifetime of its stages, respectively. Stages are processes that execute tasks concurrently/in-parallel.

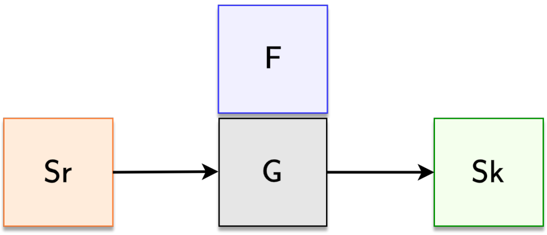


Figure 15: Initial configuration of a Dynamic Pipeline. An initial DP consists of three stages: *Source* (Sr), *Generator* (G) and *Sink* (Sk). Above the *Generator* (G) the *Filter* (F) parameter. The stages are connected through its channels, represented with the black right arrows.

These stages can be of four different types: *Source* (Sr), *Filter* (F), *Generator* (G) and *Sink* (Sk). *Source* stage are the responsible of obtaining the input data stream and feeding it into the pipeline. *Filter* stages maintain a state and process the incoming data processing it accordingly and/or passing it again to the pipeline. *Generator* stage is in charge of spawning new *Filter* stages when needed based on the incoming data, providing the *dynamic* behavior to the model. Finally, *Sink* stage receives the results, processing and acting on them as needed. Figure **??** represents the initial configuration of a DP and Figure **??** depicts the stages of the DP after a possible evolution, where the *Generator* has created two *Filter* stage instances.

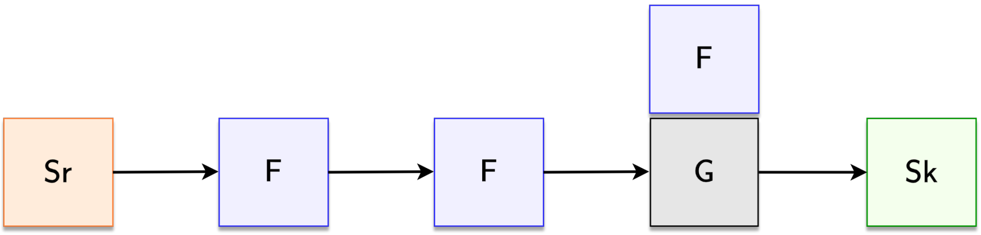


Figure 16: Evolution of a DP. After the creation of two *Filter* F stage instances of the *Filter* parameter (above the G stage) by the *Generator* G stage.

The DPP has been used to model many different problems. In [**DP-bitriangles2021**] they successfully solved the problem of progressively identify and enumerate bitriangles (i.e. a specific graph pattern) in bipartite evolving graphs. In [**DP-Lugosi˙Enes˙2019**] DPP is used to model the problem of multidimensional range queries, that is, selection queries on objects in a k-dimensional space. Finally, in [**DP-Benedi˙Garcia˙2024**] they solve the problem of computing and maintaining the minimum spanning tree of dynamic graphs.

In our case, we envision the architecture of our continuous query engine to detect anomalous ATM transactions under the DPP, where the continuous input data stream is the stream of the bank ATM transactions, and we track the activity of a Card on a certain *Filter* stage. Details on how the modeling of our problem with the DPP can be found in **??**.

3.9 Continuous Query Engine - $\mathsf{DP}_{\mathsf{ATM}}$

To explain:

- Encaje/Uso del DP para nuestro engine. Stages, qué hace cada stage (pseudocodigo/algoritmo de cada stage). Canales.
- Establecer algoritmos para identificar los patrones asociados a la busqueda de anomalias.
- Explicar la idea de que se define como para poder evaluar muchas continuous queries de forma simulatánea (pero que de momento sólo evaluamos una, para nuestro proof of concept)
- Detalles más técnicos - Implementación
  - Descripción y evaluación del lenguage usado (golang)
  - Graph-based query language
  - Tools & proper system configuration (distintas configuraciones en base al número máximo de tarjetas que contiene cada filtro...)
- Windowing?

Amalia: El windowing no lo menciones hasta las conclusiones.

In this section we define a proper architecture of a continuous query engine for detecting anomalous ATM transactions on a continuous, unbounded, input stream of card-ATM transactions/interactions. We propose an engine that is modeled following the Dynamic Pipeline Paradigm DPP (see **??**), the $\mathsf{DP}_{\mathsf{ATM}}$, where, by definition, its architectural framework gets defined as a DP .

Fernando: Encaje/Uso del DP para nuestro engine. Volatile subgraphs, cards, filters... Explicar la idea de que se define como para poder evaluar muchas continuous queries de forma simulatánea (pero que de momento sólo evaluamos una, para nuestro proof of concept)

Fernando: Subgrafos

The core idea of the $\mathsf{DP}_{\mathsf{ATM}}$ is to save and construct *volatile* subgraphs of interactions for each of the cards, with the objective to keep track of the ATM transaction/interaction activity of each of the cards of the bank system. The card subgraphs are constructed with the interactions belonging to a certain card. They are defined as *volatile* in the sense that the interactions that compose them are not intended to remain infinitely in the card subgraph, but only for a decided window of time.

Fernando: Evaluación de muchos tipos de queries de forma simultanea. Un subgrafo por cada tipo de query

These card subgraphs are the core object on which the detection queries of anomalous PG graph patterns takes place. For each card and for each continuous query pattern, a card subgraph is maintained, allowing the evaluation of many continuous different queries simultaneously. Note that, for each card, a different subgraph for each kind of continuous query is defined due to the possible many distinct time window policies for each particular kind of query.

Fernando: TODO: Poner un dibujo de un subgrafo volátil

To properly characterize the DP architecture we need to define the configuration and behavior of each of the stages as well as the channels connecting them. The different stages are connected by two communication channels: the event channel carries the interactions of the input data stream and the alert channel, which is a direct channel that connects each *Filter* stage with the *Sink* stage, carries the alerts corresponding to the different possible anomalous transaction patterns detected in a *Filter* .

Fernando: Duda channels: sólo el de alerts o el de checks. Ahora en la impl tengo checks (incluidos los alerts). Se hizo con la idea de sacar todos los resultados de los checks para ver el continuous delivery of results en los experimentos. Ahora bien, en la impl. final yo diria que solo sacaría las alerts, para tener menos overhead y que el aviso de fraude pudiera llegar antes.

Amalia: Para explicar puedes poner el de checks y que se vea que se van haciendo muchos checks y sólo algunos provocan alerts. Luego puedes explicar que en una ”working” version no se pondrías los checks para ser más eficientes y tardar menos en dar los alerts como bien dices.

Regarding the stages, the *Filter* stage is defined to be the *continuous query evaluator* for a certain subset of bank cards, maintaining the subgraphs for the cards subset that are induced by the interaction edges.

With this, the $\mathsf{DP}_{\mathsf{ATM}}$ algorithm overview is as follows: when an interaction e (with its properties' values) arrives to the $\mathsf{DP}_{\mathsf{ATM}}$, the *Source* stage Sr registers it into a standard transactional log file. Then, Sr passes e to the next stage. If there exists a *Filter* parameterized with the value of the property *number_id* of the Card vertex c that is incident to e, this *Filter* keeps e in its state, in particular adding e to the corresponding card subgraphs. Otherwise, the *Filter* passes e to the next stage. In the case of e belonging to the *Filter* , this stage, as the *continuous query evaluator* of the Card c, decides if there is a match with (some of) the continuous query pattern(s) evaluated and emits an alert to the *Sink* Sk reporting the finding. Hence, answers are the detected anomalies and they are emitted as they are obtained in filters. When answers/alerts arrive to Sk , this stage post-processes and output them. In addition, Sk maintains an answer log file. The fact that an interaction arrives to G means that there were not previous interactions having the same value of Card property *number_id* and thus, a new filter parameterized with this new value is spawned.

Fernando: TODO: Poner imagen de DP con filtros y cada filtro con los subgrafos

More detail regarding each of the stages behavior is provided next. An example of the pipeline schema of a $\mathsf{DP}_{\mathsf{ATM}}$ instance is shown in Figure **??**.

- *Source* Sr : Receives the stream of the card-ATM interactions of the bank. Each interaction e is represented as an *interaction* relation/edge of the volatile property graph model matching a Card and an ATM **??**. Sr registers incoming interaction e on a transactional log file and passes e through the event channel to the pipeline.

- *Filter* F : *Filters* are defined to be the *continuous query evaluators* for a certain subset of the bank system cards. In particular each F is defined by a subset of root parameters $V_{\mathsf{F}} = \{v_1, \dots, v_k\}$, representing the Cards being tracked by F . Therefore, each root represents a Card c where v_i is the Card property *number_id* value: $v_i = \mathsf{c}.number\_id$. Each *Filter* is defined to have a maximum capacity in terms of cards being tracked. This maximum capacity is defined by the parameter *maxFilterSize*. This limits the maximum size of the subset V_{F}, so that $|V_{\mathsf{F}}| \leq maxFilterSize$.

  With this, whenever an interaction e coming from the pipeline reaches *Filter* F , it firsts checks whether e belongs to *Filter* F . This is the case when e is incident to one of the roots of V_{F}, v_i, which has the same *number_id* property value as the Card c of the interaction e ($\mathsf{e}.number\_id$), that is when $v_i = \mathsf{e}.number\_id$. This means that F is currently tracking the activity of the Card c to which the interaction e belongs.

  Fernando: Se explica cómo se tiene hasta ahora. Pero se debería de explicar una propuesta para el caso en el que se dejara de trackear la actividad de una tarjeta / cómo sería el proceso para poder destruir un filtro/reducir el pipeline. **Hasta ahora**: If the filter is not full then we add the not belonging cards to it, until it is full. This is what is done so far, since we are not considering the case to shrink the pipeline. On which some cards activity will be stopped to being tracked.
  TODO: Definir una posible propuesta para esto en el caso de que se hiciera... relacionado con lo de la ventana...
  - Spawning case is well defined.
  - Shrinking case idea: all the cards of the filter have to be "outdated" so that the filter can be eliminated... (not done for the moment, infinite window considered)

  Another possible case on which e is decided to belong to *Filter* F , is when, although the Card c of e is not currently being tracked by F , it is decided to start doing it since F still has the capacity to track more cards: $|V_{\mathsf{F}}| < maxFilterSize$. Therefore, card c is introduced as a new root parameter $v_{k+1} = \mathsf{e}.number\_id$ to V_{F}: $V_{\mathsf{F}} = V_{\mathsf{F}} \cup v_{k+1}$. These two belonging conditions are summarized as:

$$\mathsf{e} \in \mathsf{F} \iff (\exists v_i \in V_{\mathsf{F}} \text{ such that } v_i = \mathsf{e}.number\_id) \vee (|V_{\mathsf{F}}| < maxFilterSize) \,.$$

  Otherwise ($\nexists v_i \in V_{\mathsf{F}}$ such that $v_i = \mathsf{e}.number\_id \wedge |V_{\mathsf{F}}| = maxFilterSize$), F passes e to the next stage.

  In the case of e belonging to F , F checks if there is a match with (some of) the continuous query pattern(s) evaluated and emits an alert(s) to the *Sink* Sk . For this, e will be added to the corresponding card c subgraph(s) with root $v_i = \mathsf{e}.number\_id$, and then perform the algorithm to test (each of) the continuous query pattern(s) with their associated card c subgraph(s).

  Fernando: Esto justificarlo así?

For a card c, we will store one different subgraph for each of the continuous query patterns evaluated. A different subgraph for each of the continuous query patterns is needed since the evaluation politics of each of the patterns may be different. For instance, for a specific pattern we may need to store a full list of interaction edges with some specific properties, whereas for others only the last interaction edge. Or even just some specific properties and not a subgraph of interactions. This will depend on the definition on each of the specific continuous query patterns considered.

The test of a continuous query pattern is done by means of its associated card *continuous query pattern subgraph* stored by F and the information retrieved from the stable PG to identify patterns and solve constraints. This is, indeed, the way to evaluate continuous queries.

- *Generator* G : Is the stage in charge of stretching the pipeline by spawning new *Filter* F stages when needed. In particular this is the case whenever an interaction e arrives to G . At this point this means that there was no F in the pipeline to which this interaction belonged. That is, whenever no F was tracking the activity of the Card c with property value *number_id* to which this interaction corresponds and all the running F were full of capacity in terms of the number of maximum cards *maxFilterSize* that they can track. In this case a new F is spawned, initially tracking the activity of this Card c with property value *number_id* and creating new card subgraphs with the interaction e.
- *Sink* Sk : It is in charge of receiving all the alerts coming from the *Filter* stages and to correspondingly act on them as the bank requires. This could be done in terms of communicating the alert to the corresponding cardholders, emitting a message for validating that the operation was done by the owner, freezing the corresponding cards, and so on. This will have to be defined by the corresponding bank as desired. In any case, Sk maintains an answer log file where all the emitted alerts are registered. Additionally, an event log file is maintained, to register other internal system events.

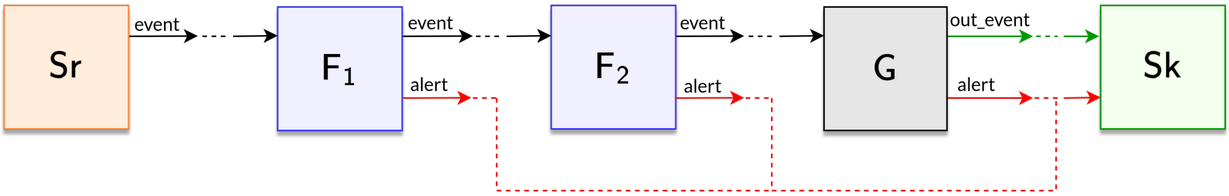


Figure 17: Example of the Pipeline Schema of a $\mathsf{DP}_{\mathsf{ATM}}$ Instance

Fernando: Windowing?

When the time interval window is over, the $\mathsf{DP}_{\mathsf{ATM}}$ is, in some sense, reset according to the given window policy. Note that the window policy must take into account stored data that might be valid in between two windows and handle the transition properly.

Fernando: Detalles técnicos - Implementación

DP$_{\text{ATM}}$ - Implementation

The implementation of the proof of concept can be found in Github[4]. It was developed using the version `1.20` of the `Golang` language. One of the main reasons why we decided to use this language is its inherent capacity to support concurrent programming [**Go-cbtnuggets˙concurrency**, **Go-medium˙concurrency**, **Go-reliasoftware˙concurrenc** which is the computing technique that we need to implement the DP schema for our DP$_{\text{ATM}}$ engine. In `Go`, concurrency is achieved primarily through `goroutines` and `channels`. `Goroutines` are lightweight, independent concurrent green threads, which are managed by the `Go` runtime scheduler, and enable concurrent execution of functions, in our case stages. The communication between `goroutines` is accomplished via different channels. This provides a safe and efficient method to pass complex data between the different `goroutines`. Unlike traditional threads, `goroutines` have a small memory footprint and can be created in large numbers with minimal overhead. The `Go` runtime includes an efficient scheduler that multiplexes `goroutines` onto CPU cores, reducing context switching overhead and optimizing resource use. However note that, `goroutines` are not inherently parallel. By default, `Go` uses only one operating system thread, regardless of the number of `goroutines`. We need to set up the `GOMAXPROCS` [5] environment variable to the number of logical processors, to achieve that the `Go` runtime scheduler multiplexes the `goroutines` onto all the possible logical processors specified.

Another advantage that made the election of `Go` quite suitable was its easy form to interact with `Neo4j`. `Go` provides the `Neo4j Go driver`[6] to easily interact with a Neo4j instance through a `Go` application. More details regarding the connection are later explained.

Neo4j Connection

Our DP$_{\text{ATM}}$ system needs a way to interact with the stable bank database PG instance in order to retrieve additional information related with the cardholders or ATMs for the evaluation of the continuous queries. The connection to the Neo4j graph database instance that represents the stable bank PG database is implemented using the version `v5.24.0` of the `Neo4j Go driver`[7]. Next we give an overview of some important details on the usage of this driver in order to connect and query the Neo4j instance. More detail on the methods we used can be found in the official driver module websites [**neo4j-go-neo4j˙go˙driver**, **neo4j-go-neo4j˙go˙manual**].

In our implementation of the DP$_{\text{ATM}}$ system in `Go` we developed the `Go` module `internal/connection` to deal with the connection management with the Neo4j stable bank graph database instance. On it we provide all the needed functions to connect to the database, create connection sessions, and to query it.

[4] `https://github.com/FCanfran/ATM-DP`

[5] `https://pkg.go.dev/runtime#GOMAXPROCS`

[6] `https://neo4j.com/docs/go-manual/current/`

[7] `https://pkg.go.dev/github.com/neo4j/neo4j-go-driver/v5@v5.24.0/neo4j`

- **Initial connection setup**: The DP_{ATM} system initially sets up the connection through the creation of a `DriverWithContext` object.
  In the `internal/connection` module `SafeConnect()` is the function that we implement to construct the `DriverWithContext` object and verifies that a working connection can be established through the `.VerifyConnectivity()` method. The `DriverWithContext` object holds the details required to establish connections with a Neo4j database, allowing connections and creation of sessions. It is a sharable object among threads. To provide the required details we used the module package `godotenv`[8] to obtain the *URI* and the credentials from a `.env` file, where the related environment variables are specified. For the connection with our Neo4j instance we use the `Bolt`[9] application protocol, which is the protocol used for interaction between Neo4j instances and drivers. It listens on port 7687 by default. To illustrate, we show an example of a `.env` file, from which the connection credentials will be gathered by our application. On it the connection details of a toy local Neo4j instance are shared:

```
NEO4J_URI="bolt://localhost:7687"
NEO4J_USERNAME="neo4j"
NEO4J_PASSWORD="xxxxx"
```

Listing 18: Example of a `.env` file, from which from which the connection credentials will be gathered by our DP_{ATM} application.

- **Connection sessions**: `Sessions` act as concrete query channels between the driver(`DriverWithContext` object) and the server. We need them in order to be able to run `Cypher` queries from each of the working *Filter* stages. Each *Filter* stage creates a different `session` to interact with the Neo4j database.

  In the `internal/connection` module, the functions `CreateSession(...)` and `CloseSession(...)` are the functions for the creation and closing of a `session`. They are created from the `DriverWithContext` object.

- **Running queries**: We provide the methods `ReadQuery(...)` and `WriteQuery(...)`, which create a *managed transaction* to retrieve data from the database or alter it, respectively, through a `Cypher` statement. Internally they call the `ExecuteRead(...)` and the `ExecuteWrite(...)` functions of the `Neo4j Go Driver`, which execute the given unit of work in a read/write transaction, via the provided session.

  In the DP_{ATM} we make use of the `ReadQuery(...)` function from each of the *Filter* stages, using its own `session`, to query the database using `Cypher` statements.

Fernando: DUDA: Pongo una referencia a esto?

Neo4j limits the number of parallel transactions to 1000 by default. However, so far no reference regarding the limit on the number of parallel sessions has been found.

[8] `https://pkg.go.dev/github.com/joho/godotenv`

[9] `https://neo4j.com/docs/bolt/current/bolt/`

In any case, it is important to remark that many multiple parallel sessions may cause overhead to the database. This can be the case for the architectural design we are proposing, where we have a session per each *Filter* stage.

Communication

The communication of the different stages is carried via different Go channels. In general, although otherwise specified all the channels that we use are buffered channels of size 5000. All the channels are described next:

- event : event channel. Its main purpose is to carry the interaction edges across the DP stages, from Sr to G , passing by all the F ’s. The event data type consists on a EventType label and in a Edge object. The EventType label indicates the type of event that can be either: EdgeStart representing an opening of an interaction, EdgeEnd representing an interaction closing, EOF, representing the *End Of File* event so that the DP can become to an end, finalizing all the stages, and the LOG event for internal log messages of the system.

```
type Event struct {
Type      EventType
E         Edge
}
```

Listing 19: Event Data Type

Edge is the data type that we defined for the interaction edges. This object will be relevant in the case of the EdgeStart and EdgeEnd events, since in this case the Edge object will be containing the information of the opening and closing interaction, respectively, as defined in the volatile property graph data model definition (see ??).

```
type Edge struct {
Number_id string    // Card id
ATM_id    string    // ATM id
Tx_id     int32     // id
Tx_type   TxType    // type (withdrawal/deposit/inquiry/
transfer)
Tx_start  time.Time // start datetime(DD/MM/YYYY HH:MM:SS)
Tx_end    time.Time // end datetime(DD/MM/YYYY HH:MM:SS)
Tx_amount float32   // amount
}
```

Listing 20: Data Type for the interaction edges in Go

where TxType is a custom type made for the different interaction types: withdrawal, withdrawal, withdrawal and withdrawal.

```go
type TxType uint8
const (
 Withdrawal TxType = 0
 Deposit           = 1
 Inquiry           = 2
 Transfer          = 3
 Other             = 4
)
```

Listing 21: `TxType` Data Type, for the different interaction types

- alert : `alert` channel. It carries the alerts corresponding to the different possible anomalous transaction patterns detected in a *Filter* . It is a channel directly connecting each of the *Filters* with the Sink stage. This means that when an alert is emitted it does not have to travel through all the remaining F stages of the DP nor the G stage, allowing a faster communication of the alert to the Sk stage, in charge of processing the alerts. The `alert` data type consists on a `Label` to indicate the type of fraud pattern to which it corresponds, an `Info` string to indicate additional information related with the alert, and finally `Subgraph`, the subgraph data structure that triggered the alert. Note that this subgraph does not need to be the full subgraph of the card that triggered the alert, it can be just the part of it involved in the alert. For instance, in the case of the fraud pattern I, these subgraph is composed of the two interaction edges that caused the trigger of this kind of fraud pattern alert.

  ```go
  type Alert struct {
   Label    string
   Info     string
   Subgraph Graph
  }
  ```

  Listing 22: `Alert` Data Type

- out_event: direct dedicated `event` channel between the G and Sk .
- internal_edge: Internal `event` channel between a F stage and its related FW substage. It only pass interaction `Edge` events (`EdgeStart` and `EdgeEnd`), that have been determined to belong to the *Filter* , so that the related FW of F can do the corresponding processing with it.

Stages

In what follows we give specific implementation details of each of the stages of the DP paradigm used for the $\mathsf{DP}_{\mathsf{ATM}}$. Each stage takes the form of a `goroutine` of the `Go` language.

Source

Source stage is designed to be the connection point of the $\mathsf{DP}_{\mathsf{ATM}}$ with the bank ATM network to receive the interactions produced on these ATMs, which compose the input interaction stream.

Fernando: DUDA: Aqui esto no se como ponerlo... - Decir algo de kafka message queue (como en seraph)?

In a real-case scenario, these interaction events could be sent by the ATMs of the bank network and be received by a message queue on our $\mathsf{DP}_{\mathsf{ATM}}$ system. For our proof of concept, where we generated our own synthetic stream of transactions in a `csv` file, the interactions are read from these files, parsed into `Edge` data types and provided to the pipeline in different ways depending on the kind of simulation we perform. As it will be shown in the Experiments section, we implemented two different cases of simulations. The real-case scenario and the high loaded test scenario. In the first case, the interactions, although read by a file of artificial simulated interactions, are provided to the pipeline data stream in such a way that they simulate their actual arrival time to the system, with the corresponding time separation between them. In the second case, the interactions are provided just one after the other as fast as possible as they are read.

In any case, we want the reading of the input file to be the fastest possible, so to minimize the potential bottleneck derived from the operation of reading a file, we utilized a buffered reader of the `bufio` package, which reads chunks of data into memory, providing buffered access to the file. This buffered reader was provided to a `csv` reader of the `encoding/csv` package to read the buffered stream as `csv` records.

```
reader := csv.NewReader(bufio.NewReader(file))
```

Listing 23: `csv-bufio` reader

Another optimization that was done in order to be able to minimize this bottleneck on the reading of the interactions from the `csv` file, was reading by chunks the `csv` records/rows. In particular, this was done by having a *worker* subprocess, implemented as an anonymous `goroutine` inside Sr , whose task was to continuously read records from the file using the `csv-bufio` reader accumulating them in a chunk of rows that were provided through a channel to Sr whenever they reached the defined *chunkSize*. These records were read directly as `string` data types. On its side, whenever Sr receives a chunk of rows, it takes each of the rows on it, parses it to the `Edge` data type and sends it through the pipeline to the next stage.

The *chunkSize* was selected to be of 10^2 rows. In **??** we provide an experimental analysis that proves and justifies the benefits of this buffered and chunked file reading. On it the `encoding/csv` package performance is compared to other variants using the `apache/arrow` package with different combinations of *chunkSize*. We also analyze the benefits of introducing the *worker* subprocess to perform the chunked reading.

Fernando: Explicar más en detalle como va lo de la lectura por chunks con el worker?

Filter

Regarding the *Filter* stage, there are four main aspects worthy to describe on its implementation: the card subgraph data structure implementation, the *decoupled event handling* implementation, the *Filter* multiple cards support and the continuous query evaluation.

- **Card Subgraph Data Structure**
  It contains the interaction edges related with the continuous query fraud pattern evaluation of a specific card. The card subgraph data structure is selected based on the fraud patterns considered so far. Although for the implementation of the detection of the fraud pattern I (see **??**) we only need to store the last/most recent interaction for each card, we propose a `Go list` of `Edge` (interaction edges) objects as a more general data structure to store incoming interactions ordered by timestamp. This decision responds to what we see it is a more general data structure ready for its usage when implementing the detection of other kinds of fraud patterns.

  ```
  type Graph struct {
   edges *list.List
  }
  ```

  Listing 24: Card Subgraph Data Structure in Go

  A card subgraph is constructed based on the belonging interaction edges that arrive in the form of incoming `EdgeStart` and `EdgeEnd` events. `EdgeStart` event produces the creation of a new `Edge` on the data structure, whereas a `EdgeEnd` event completeS the values of the properties of the corresponding `Edge` on the data structure, that was previously created by the `EdgeStart` event corresponding to that same ATM-Card interaction.

  Fernando: TODO: Poner dibujito de subgrafos dentro del filtro?

- **Decoupled Event Handling**
  We implement a *Decoupled Event Handling*, as in the DP implemented on [**DP-Benedi˙Garcia˙2024**]. As the authors mention in this work, there exists a potential inefficiency in the way that the *Filter* F stage is defined to work: if an event belongs to the filter then it does the processing related to it. Until F does not complete this processing the events that arrive to it are not being able to be handled, even if the event does not belong to F . We have the same

scenario for the DP of the DP$_{ATM}$, where the events that can belong to F are the interaction edge `e` events.

To avoid this bottleneck on the flow of interaction edge `e` events, the *Filter Worker* FW is introduced as a substage of the *Filter* F stage. F and its associated FW will be running in different `goroutines`. The idea is that F acts as a dedicated *mailbox*, reading the events coming from the `event` channel and *filtering* them, that is, deciding whether an incoming interaction edge `e` belongs to F or not. In the case `e` belongs to F , F forwards `e` to the FW , otherwise it continues passing `e` through the pipeline. FW is dedicated to do the corresponding processing with the interaction edges belonging to the stage that are forwarded by F .

This decoupling allows to do the processing of belonging interaction edges while not blocking the pipeline event flow, since at the same time it does the *filtering* of events.

In our `Go` implementation, we decided to implement FW as an internal anonymous `goroutine` of the F `goroutine`, instead of an external (named) `goroutine`.

To communicate the interaction edges between F and FW we use an internal channel internal_edge . Another possible option considered was a shared buffer of interaction edges. In this last case a mutex would have been needed, since F and FW can possible write and read, respectively, into this buffer at the same time. The *mutex* is needed to avoid race conditions in the sharing of the buffer. However, channels are a much more simple alternative to lead with this communication, as they are specifically designed for synchronization and passing the ownership of data, which is the case we are dealing with. Some references on the preference usage of channels over mutex [**Go-channels-mutex-gowiki˙mutex˙channel**, **Go-channels-mutex-stackoverflow˙mute**

The decision to implement FW as an internal anonymous `goroutine` also provided a way to simplify the code, since FW can access the variables of the scope of F (no need to pass them as parameters). This is particularly useful in the case of the alert channel, to which FW is able to write directly. Same in the case of the internal_edge channel.

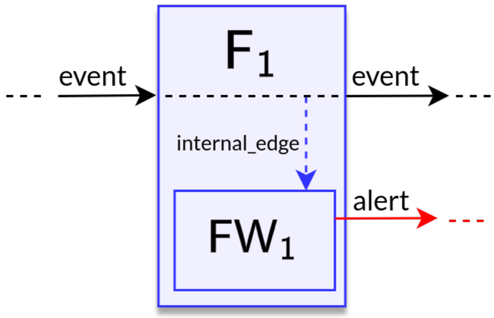


Figure 18: Filter Worker detail

- **Multiple Cards Support**
  In an initial first toy implementation, we were tracking the activity of only one card per *Filter* , meaning that for each bank card we were dedicating a `goroutine` on the form of a *Filter* stage. This was done as a way to get started, it fast became obvious that this is was an unnecessary waste of computational resources. On a real case scenario, where the number of ATM-card interactions a bank card in a day is not expected to be higher than one on average

  Fernando: TODO: Poner una referencia formal

  , it became clear the need of allowing to share each *Filter* stage for multiple bank cards.

  Formally, as described on **??**, each *Filter* F was implemented to be able to hold a certain subset of root parameters $V_{\mathsf{F}} = \{v_1, \ldots, v_k\}$, representing the Cards being tracked by F .

  In `Go` to achieve the support of multiple cards per *Filter* we decided to use hash tables. Go provides a built-in `map` type that implements a hash table[10]. We used two different maps, using the card id of the interaction edge e.*number_id* as the key of both maps.

  - `cardList map`: It is used by F to determine whether an interaction edge e belongs to F , $\mathsf{e} \in \mathsf{F}$. Only accessed by F .
  - `cardSubgraph map`: To index the interaction edge e to the corresponding card subgraph. Only accessed by FW . Note that, for our proof of concept, as we are considering only one fraud pattern, we only need one `cardSubgraph map` data structure. However, note that more will be needed if we consider more fraud patterns with different card subgraphs.

  ```
   var cardList map[string]bool = make(map[string]bool)
   var cardSubgraph map[string]*cmn.Graph = make(map[string]*
  cmn.Graph)
  ```

  Listing 25: Hash Tables `map` Data Structures on a *Filter* F stage

  One could think of using one single `map` data structure to do the check $\mathsf{e} \in \mathsf{F}$ and at the same time index e to the corresponding card subgraph, if it is the case. However, the reason why we need two `maps` and not only one is to respect the architecture of the decoupled event handling. On it, we have two `goroutines` F and FW (as an anonymous internal `goroutine` of F) dedicated to check $\mathsf{e} \in \mathsf{F}$ and to do the processing of e, respectively. Having only one `map` data structure, would mean that both F and FW would be doing simultaneous read/write operations on the `map`, which is unsafe as is

[10] https://go.dev/blog/maps

not defined what happens (possible race conditions) in `Go` in this situation. The runtime also warned us to avoid this by alerting us with a fatal error message if we try to share this data structure without any kind of synchronization tool: *Issue: "fatal error: concurrent map read and map write"*. Therefore a *mutex* or a concurrent map implementation like a `syncmap` would be needed to control the concurrent access to this shared data structure. For simplicity we decided to avoid sharing the data structure and dedicating one `map` for each of the F and FW stages.

- **Continuous query evaluation**

  As already mentioned, the continuous query evaluation of the different fraud patterns for the cards is accomplished in the *Filter* stage. For a card `c` this is achieved with the evaluation of the algorithms that characterize each of the defined fraud patterns. These algorithms are evaluated over the corresponding card subgraphs and the information retrieved from the stable PG bank database, identifying if there is a subgraph that matches the given patterns and satisfies its constraints.

  So far, we implemented the fraud pattern I, related with the characterization of a card cloning scenario, as defined in the algorithmic description **??**. In **??** we provide a more detailed description of the implementation of this algorithm. Whenever a `EdgeStart` event arrives to FW through the internal_edge channel the checking algorithm `checkFraud` is performed. `checkFraud` is executed over a non-empty card subgraph S_c and the interaction `Edge` e_{new} of the `EdgeStart` event. It is checked that there exists a sufficient time distance between e_{new} and e_{last}; the previous added edge to Sc before e_{new}.

  To do it, we need to obtain the minimum needed time `t_min` to traverse the distance between the ATMs of the e_{new} and e_{last} interactions: ATM_{last} and ATM_{new}, corresponding to the ATMs with identifiers e_{last}.`ATM_id` and e_{new}.`ATM_id`, respectively. `t_min` is obtained through the function call obtainTmin(e_{last}, e_{new}) on line **??**. This function obtains the location coordinates of the two ATMs through two `Cypher` query to the Neo4j stable bank database, using the `ATM_id` identifiers (see code listing **??**). This is needed since, by definition interaction edges do not contain this information, and therefore the stable bank database needs to be queried to retrieve it, so to be able to do the check of this graph pattern. This query is executed using the function `ReadQuery` from the `internal/connection` module of our DP_{ATM} implementation.

Algorithm 5 `checkFraud`($\mathsf{S_c}, \mathsf{e_{new}}$)

```
Require: S_c is a non-empty subgraph of interaction edges of card c, e_new is the Edge
    related with the new incoming opening interaction EdgeStart of card c
1:  $e_{last} \leftarrow S_c[|S_c| - 1]$ {Retrieve the last edge from the subgraph $S_c$}
2:  if $e_{last}$.Tx_end is empty then
3:      LOG: Warning:  A tx ($e_{new}$) starts before the previous ($e_{last}$) ends!
4:      return
5:  end if
6:  if $e_{last}$.ATM_id $\neq$ $e_{new}$.ATM_id then
7:      t_min $\leftarrow$ obtainTmin($e_{last}, e_{new}$)
8:      t_diff $\leftarrow$ $e_{new}$.Tx_start $-$ $e_{last}$.Tx_end
9:      if t_diff $<$ t_min then
10:         emitAlert($e_{last}, e_{new}$)
11:     end if
12: end if
```

```
getATMLocationQuery :=
MATCH (a:ATM) WHERE a.ATM_id = $ATM_id RETURN
a.loc_latitude AS loc_latitude,
a.loc_longitude AS loc_longitude
```

Listing 26: Code of the constructed `Cypher` query in `Go` to obtain the location coordinates of an ATM with its id on the Neo4j graph database

Once the location coordinates of both ATMs are obtained from the query:

- $\mathsf{coords_{last}} = (\mathsf{ATM_{last}}.\texttt{loc\_latitude}, \mathsf{ATM_{last}}.\texttt{loc\_logitude})$
- $\mathsf{coords_{new}} = (\mathsf{ATM_{new}}.\texttt{loc\_latitude}, \mathsf{ATM_{new}}.\texttt{loc\_logitude})$

then `t_min` is calculated as: $\texttt{t\_min} = \texttt{distance}(\mathsf{coords_{last}}, \mathsf{coords_{new}}) / \texttt{maxSpeed}$. $\texttt{distance}(\mathsf{coords_{last}}, \mathsf{coords_{new}})$ is the great-circle distance or orthodromic distance between the two coordinate points. It is calculated using the `Go` haversine package[11].

`maxSpeed` is a parametrizable constant indicating the maximum speed at which it is assumed that the distance between any two geographical points can be traveled. So far we defined it to be `maxSpeed` = 500 km/h. This parameter will have to be defined by the bank system. Of course, a more refined version could calculate this minimum time considering more variables, so to provide a more precise calculation.

If the required conditions hold, then an `Alert` is emitted through the alert channel to the *Sink* stage. This is summarized on the function emitAlert($\mathsf{e_{last}}, \mathsf{e_{new}}$) on line **??**. The `Alert` will contain the subgraph built with the two interaction edges causing the alert: $\mathsf{e_{new}}$ and $\mathsf{e_{last}}$ on the `Subgraph` field, as well as the indication on the type of fraud on its `Label` field as the type ”1” and some optional additional information related with the anomaly on `Info`. A possible

[11] `https://github.com/umahmood/haversine`

implementation of this function is shown in `Go` in the **??** code listing. Note that out_alert refers to the output alert channel that connects the *Filter* stage with the *Sink* stage (`chan<- cmn.Alert`).

```go
var fraudAlert Alert
subgraph := NewGraph()
subgraph.AddEdge(last_e)
subgraph.AddEdge(new_e)
fraudAlert.Label = "1"
fraudAlert.Info = "fraud␣pattern"
fraudAlert.Subgraph = *subgraph
...
out_alert <- fraudAlert
```

Listing 27: Possible implementation of emitAlert($\mathsf{e}_{\mathsf{last}}, \mathsf{e}_{\mathsf{new}}$)

With all these ingredients a final `Go` implementation of the *Filter* is provided on the code listing **??**.

```
func filter(event cmn.Event, in_event <-chan cmn.Event, out_event
   chan<- cmn.Event, out_alert chan<- cmn.Alert) {

var edge cmn.Edge = event.E
var cardList map[string]bool = make(map[string]bool)
var cardSubgraph map[string]*cmn.Graph = make(map[string]*cmn.Graph
   )
cardList[edge.Number_id] = true
internal_edge := make(chan cmn.Event, cmn.ChannelSize)
// termination synchronization channel F-FW
endchan := make(chan struct{})

context := context.Background()
session := connection.CreateSession(context)
defer connection.CloseSession(context, session)

// FW
go func() {
 // auxiliary subgraph variable
 var subgraph *cmn.Graph
 cardSubgraph[edge.Number_id] = cmn.NewGraph()
 subgraph, _ := cardSubgraph[edge.Number_id]
 subgraph.AddEdge(edge)
Worker_Loop:
 for {
    event_worker, _ := <-internal_edge
    switch event_worker.Type {
      case cmn.EOF:
      // finish the worker
      endchan <- struct{}{}
      break Worker_Loop
    case cmn.EdgeStart:
      subgraph, ok = cardSubgraph[event_worker.E.Number_id]
      if !ok {
        // first edge related to the card on subgraph
        cardSubgraph[event_worker.E.Number_id] = cmn.NewGraph()
        subgraph, _ = cardSubgraph[event_worker.E.Number_id]
        subgraph.AddEdge(event_worker.E)
      } else {
        // already an edge of the card
        isFraud, alert := subgraph.CheckFraud(context, session,
   event_worker.E)
        if isFraud {
          alert.LastEventTimestamp = event_worker.Timestamp
          out_alert <- alert
        }
        // set as new head of the subgraph (only save the last edge
   )
        subgraph.NewHead(event_worker.E)
      }
    case cmn.EdgeEnd:
      subgraph, ok = cardSubgraph[event_worker.E.Number_id]
        if !ok {
          cardSubgraph[event_worker.E.Number_id] = cmn.NewGraph()
          subgraph, _ = cardSubgraph[event_worker.E.Number_id]
          subgraph.AddEdge(event_worker.E)
        } else {
          subgraph.CompleteEdge(event_worker.E)
        }
    }
    }
}()

// Filter
Filter_Loop:
 for {
    event, _ := <-in_event
    switch event.Type {
    case cmn.EOF:
        internal_edge <- event
        <-endchan
        out_event <- event
        break Filter_Loop
    case cmn.EdgeStart, cmn.EdgeEnd:
        if cardList[event.E.Number_id] {
            internal_edge <- event
        } else if len(cardList) < cmn.MaxFilterSize {
            cardList[event.E.Number_id] = true
            internal_edge <- event
        } else {
            out_event <- event
        }
    }
}
close(internal_edge)
close(out_event)
}
```

Listing 28: A *Filter* stage Go implementation

Generator

The *Generator* G main functionality is spawning a new *Filter* F stage whenever it receives an interaction Edge event. This event is provided as a parameter to the new spawned F so that it can be then processed there.
Whenever the new F is spawned, the pipeline is reconnected accordingly: F takes as event input channel the original event input channel of G , and G generates a new event input channel which is set up as the output event channel for F and as the new input event channel for G . The alert is also provided to F so that it can utilize as an output channel to send alerts to the *Sink* stage.

Fernando: Poner extracto código para enseñar mejor?

Sink

The *Sink* Sk stage is in charge of reading from the event and alert . Its main functionality is to receive the alerts coming from the alert and post-processing them accordingly to the bank requirements. In our case, we just write them into an output file to record them. In addition we also maintain a log event output file with the received events received from the `event` channel.

Fernando: Actualizar los dibujos correspondientes. Poner estos y otros similares a los de la presentación del AMW2024 pero actualizados a este formato.

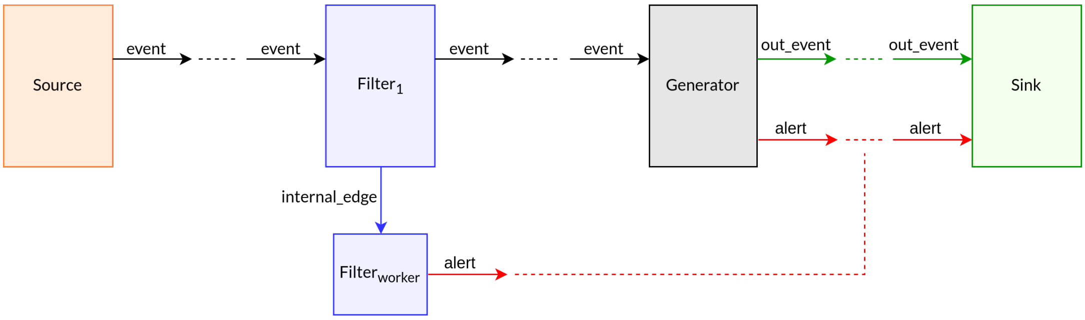


Figure 19: Pipeline Schema with Filter detail

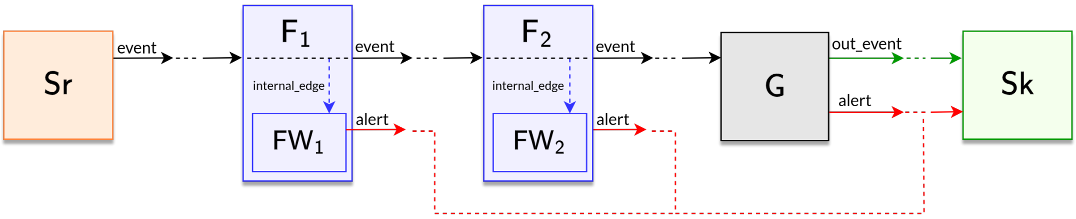


Figure 20: Pipeline Schema with Filter detail 1

4 Experimental Evaluation

TODO: Explicación de los experimentos: Introducción: qué contiene la sección

- Explicación inicial de lo que queremos probar
- Cómo se va a probar:
  - Evaluation metrics: diefficiency
  - Different bank sizes
  - Different streams
  - Scaling for the NRT tests
  - Measurement of all the checks, not only the alerts
- Resultados

In this section we detail the experimental evaluation done in order to show the performance of the $\mathsf{DP}_{\mathsf{ATM}}$ system. With these experiments we intend to show the suitability of the dynamic pipeline computational model as a real-time system capable of emitting results as they are computed, in a progressive way.

To evaluate the $\mathsf{DP}_{\mathsf{ATM}}$ as a real-time system, we conducted two kinds of experiments. On the one hand the analysis of the $\mathsf{DP}_{\mathsf{ATM}}$ on a high-load stress scenario (E1), where we study the behavior of the system in a worst case scenario, receiving a high loaded transaction stream. On the other hand, the evaluation of the $\mathsf{DP}_{\mathsf{ATM}}$ in scenarios reflecting more real-world possible conditions, with a more realistic transaction frequency (E2).

These scenarios were tested for different bank sizes, representing different possible bank types in terms of the number of cards and ATMs, as well as for different streams of transactions, representing different interval sizes of simulated card-ATM activity.

Finally, additional experiments are included such that the one regarding the description of the comparison of the different tested ways to do the reading of the stream input by the Sr stage.

Fernando: Explicar que se empezaron a hacer los dos experimentos, pero que al hacer E2 se acabó viendo que realmente se estaba tendiendo a hacer el E1, y que por eso el principal experimento de referencia fue la prueba del sistema en un high-load scenario E1.

The frequency of the transactions on a transaction stream depends mostly on the number of clients and therefore cards of the bank system.

4.1 E1: Evaluation in a High-Load Stress Scenario

In this section we evaluate the behavior of the $\mathsf{DP}_{\mathsf{ATM}}$ in worst-cases scenarios in terms of the frequency of the transactions of the input stream. This intends to prove the performance of the $\mathsf{DP}_{\mathsf{ATM}}$ in a stress scenario where the stream of transactions arriving to the system is at its maximum peak.

To evaluate the behavior of the system under these conditions, we decided to analyze not only classical real-time system metrics but also newly proposed metrics for assessing the system's continuous behavior. The selection of the metrics for the system evaluation is described in **??**.

The system will be tested for different possible bank sizes, with varying numbers of cards, ATMs, and stream sizes, simulating various time intervals of card-ATM interactions. The selection of the different possible bank sizes and the details of the creation of the synthetic streams of transactions are given in **??**. In each of the cases we intend to show the performance of the system under different possible configurations in terms of the number of filters and available cores on the running machine.

Fernando: TODO: Describir que podemos medir checks o solo alerts. Esto dónde ponerlo?

What we consider a result / an answer can variate. In principle, attending only to our system description, we consider a result to be equivalent to an alert, caused by a positive fraud check on a *Filter* stage. However, for experimental purposes, in order to show the continuous production of results by our $\mathsf{DP}_{\mathsf{ATM}}$, we will consider all the fraud pattern checks as results, even if they are negative, that is, when they do not derive in the creation of an alert.
Note that considering all the fraud pattern checks as results is done only due to experimental purposes, in order to better analyze the continuous fraud pattern checking that the system is undergoing. Note that, considering all the checks as results derives in a communication overhead between the *Filter* F stages and the *Sink* Sk stage, where the results are gathered. However, on a production version of the $\mathsf{DP}_{\mathsf{ATM}}$, alerts will be the only result considered, as there would be no interest in sending a negative fraud pattern check to Sk . Only the positive fraud pattern checks, the alerts, are of interest in that case, therefore reducing at maximum the overhead in the communication channel between the F's and Sk and the corresponding processing of results in Sk .

4.1.1 Evaluation Metrics

The $\mathsf{DP}_{\mathsf{ATM}}$ is an intended to be real-time system for the detection of card-ATM anomalous interactions. As such its evaluation must capture classical metrics such as: mean response time of an answer/detection to be produced, throughput in terms of answers emitted per unit of time, the execution time to process a full stream. Additionally, we include other kinds of metrics to quantify and evaluate the efficiency of the system over a certain time period, the so-called *diefficiency* metrics, introduced in [**exps-diefficiency**]. Unlike the classical metrics, these metrics allow us to have a more complete picture on how the system is behaving during a given period of time, and not just a reduced final picture, by evaluating the progressive emission of results in that time period.

- **Number of results: `checks` and `alerts` produced**
  Counters of the number of results that the system produces. Results can be either be counted only as alerts (positive fraud patterns), as fraud pattern checks, both positive (alerts) and negative, or as both at the same time, depending on the experiment.

- **Response Time (`RT`) and Mean Response Time (`MRT`)**
  `RT` captures the time it takes for the system to emit a result. It is the elapsed time from the moment an interaction arrives to the system until the result of its respective fraud pattern check is produced. A fine-grained description on how this metric was captured in the $\mathsf{DP}_{\mathsf{ATM}}$ system is included in **??**. The `MRT` is the average response time metric for all the results emitted by the system.

- **Execution Time (`ET`)**
  It measures the total time (in seconds) that it takes for the system to consume/process a full input stream.

- **Throughput (`T`)**
  It measures the number of results emitted per time unit. It is calculated as number of results divided by `ET`.

- **Interactions per second (`interactions/s`)**
  It measures the number of interactions that the system is able to process per time unit. It is calculated as the total number of interactions that arrived to the system divided by `ET`.

- **Time to produce the First Tuple (`TFFT`)**
  It is the time required by the system to produce/emit the first result.

- **`dief@t` and `dief@k` metrics**
  They measure the *diefficiency* - the continuous efficiency of an engine over a certain time period in terms of the emission of results - and, as mentioned allow us to have a more complete picture on how the system is behaving during a given period of time, and not just a reduced final picture. `dief@t` measures the diefficiency of the engine while producing results in the first t time units of execution time. The higher the value of the `dief@t` metric, the better the continuous behavior. `dief@k` metric measures the diefficiency of the engine while producing the first k results, after the first result is produced. The lower the value of the `dief@k` metric, the better the continuous behavior.

- **Answer Trace**
  We provide a similar definition as the one in [**exps-diefficiency**]. An answer trace can be formally defined as a sequence of pairs $(t_1, r_1), ..., (t_n, r_n)$, where r_i is the *ith* result produced by the engine and t_i is the timestamp that indicates the point in time when r_i is produced. We will record an answer trace for each of the experimental evaluations of the engine, since they provide valuable insights about the continuous efficiency - diefficiency.

To obtain the `dief@t` and `dief@k` metrics we used the `diefpy` tool [**exps-diefpy-tool**], which calculates them from the generated answer/result traces by our system. This tool provides us with other utilities for the visualization of the metrics on the obtained sets of results such as: the visualization of answer traces, the generation of radar plots to compare the `dief@t` with the other conventional/classical metrics, the generation of radar plots to compare the `dief@k` at different answer completeness percentages, among others.
We additionally extended/modified the tool for our specific needs. This was done in order to visualize other metrics, especially the `mrt`, but also others like the thoughtput `T`, `interactions/s` or the `TFFT`.

Fernando: TODO: explicar mejor la utilidad de medir dieft and diefk... resultados... answer trace si es muy bueno

4.1.2 Bank and Stream Configurations

Testing a bank system application like the $\mathsf{DP}_{\mathsf{ATM}}$ system implies deciding on different representative bank sizes and different stream sizes on which to perform the evaluation of the system. Regarding the stream of transactions the proportion of regular vs anomalous transactions also needs to be decided.

We did a brief investigation of some related works, regarding the experimental stream sizes as well as the ratio on the anomalous fraud transactions they utilize. On [**exps-atmfrauddetectionstreamdata**] the authors propose an ATM fraud detection system based on ML models where they experiment with a transaction stream of a size close to 10^6 consisting of a ratio of 0.88 regular operations to 0.12 fraudulent operations. In [**exps-costsensitivepayment**] they propose a ML algorithm based on dynamic random forests and k-nearest neighbors, and they test it with a real bank transaction stream of $\sim 5 \times 10^4$, representing the card activity of 415 different cards, with a fraudulent transaction ratio of 0.07. Finally in [**exps-featureengineering**] they evaluate the impact of their proposed feature extraction techniques for credit card fraud detection on different ML and data mining state-of-the-art models. For their experiments they used a real European transaction dataset containing $\sim 120 \times 10^6$ transactions representing a 18 months time interval, in which only ~ 40000, that is a 0.025%, are fraudulent. They also consider a smaller subset of $\sim 2 \times 10^5$ transactions with a fraud ratio of 1.5%.

Fernando: TODO: Poner mejor

Based on these references we constructed different streams with sizes and fraud ratio similar to the ones mentioned, using our generator of synthetic transactions, generated from different bank databases. The variation on the size of the simulated stream of transaction on which to test our system is not the unique we do, we also variate the bank database size in terms of the number of cards and ATMs it contains. This variation, apart from having an influence on the volume of generated transactions on a time interval when using our synthetic transaction stream generator, it is expected to have an influence on the performance of the system due to the expected added overhead when querying the Neo4j stable graph database. In our experiments we propose two bank database sizes; one small bank database with

2000 cards and 50 ATMs, and one large bank database with 500000 cards and 1000 ATMs.

Fernando: Poner referencias a bancos reales de tamaños similares?

When generating the streams, in order to obtain the desired stream sizes, we needed 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 ~ 1 transaction per day. (TO CHECK to give the exact number). This means that for each of the bank sizes being tested, in order to achieve the desired stream size, we need to decrease/increase the size of the time interval to be simulated.

Fernando: TODO: Sacar métricas exactas de cuál es el numero medio de ops por cliente de wisabi para poder decir cuál es el número de tx por dia approx que generamos con nuestro generador para poder relacionarlo

Fernando: Pongo esto?: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.

4.2 E2: Evaluation in a Real-World Stress Scenario

- E1: Mean Response Time
  - Q1: How much it takes for the system to emit the alerts from the moment the anomalous transaction was produced.
  - R1: Real time simulation. Measuring the mean response time from the start of the transaction of the detected anomalous scenario until the alert is emitted by the system.
- E2: Continuous delivery of results in a high loaded scenario
  - Q2: Behavior of the different system variations on a high loaded transaction stream scenario.
  - R2: Not real time simulation. Direct transaction stream input supply. Comparison of the continuous delivery of results of the different systems variations (diefficiency metrics).

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).

4.2.1 E1: Mean Response Time

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.

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.

Amalia: Esto tiene que ir al principio de la sección de experimentos, tienes que explicar ”con palabras” qué es lo que quieres probra y luego cómo lo haces.

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**

4.2.2 E2: Continuous delivery of results in a high loaded scenario

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).

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.*)

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 **??**).

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)—

4.2.3 Initial experiments

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 6: Different experimental setups results

Some nomenclature:

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

Some results:

1h scaling

Amalia: Los captions de las figuras tienen que ser autocontenidos.

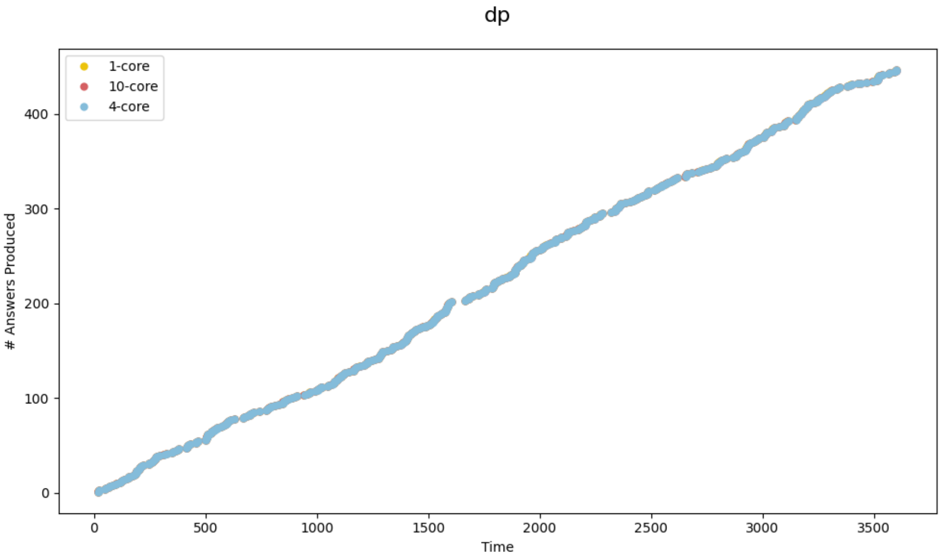


Figure 21: Trace 1h

Only for the first 10 results (alerts):

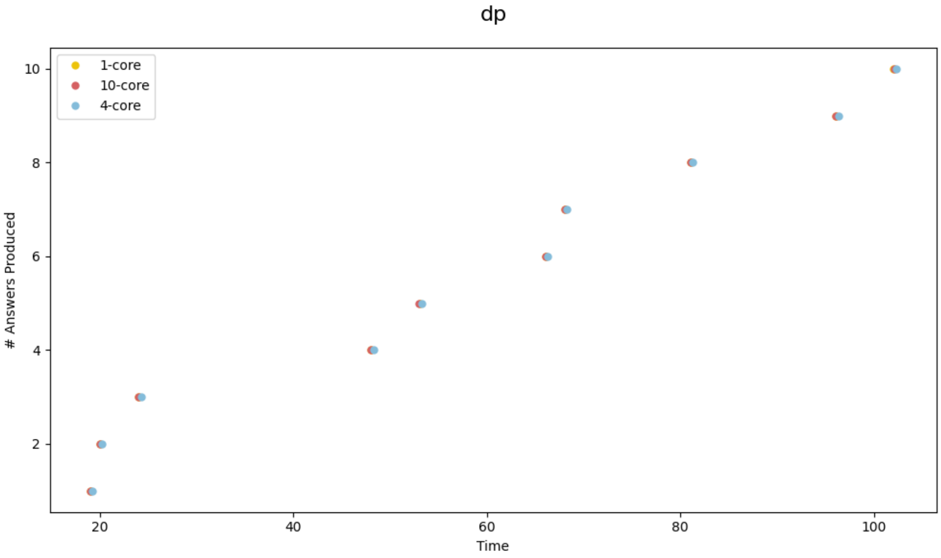


Figure 22: Trace 1h - first 10 alerts

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.

4.3 Definitive Experiments

→ We set the worker stream reading by chunks, with chunk size of 10^2.

The setups of the experiments that we are going to do are:

- Fix different bank sizes (small, $+\frac{1}{4}$, $+\frac{2}{4}$, $+\frac{3}{4}$, the biggest possible size).
  - For each, generate different stream sizes.
    - Compare different system variations in the number of cores and number of filters.

Bank Size	# Cards	# ATMs	Stream Size (# tx)
Small	2000	50 (45 internal - 5 interbank)	39959 - small
Small	2000	50 (45 internal - 5 interbank)	80744 - medium
Small	2000	50 (45 internal - 5 interbank)	160750 - big
Medium	500000	1000 (900 internal - 100 interbank)	
Big			
Big			
Big			

4.3.1 Bank size: Initial - Small

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

Transaction stream - small

- `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%.

For different core variations, we are going to try different combinations of the system in terms of the number of the maximum number of cards per filter, that consequently will produce an inverse variation in the number of filters of the system.

# cards per filter	# filters
2000	1
1000	2
400	5
200	10
100	20
50	40
20	100
10	200
4	500
2	1000
1	2000

- # of times / runs each job = 10.
- Maximum RAM limited to 16GB.

4.4 E1: Mean Response Time

- Q1: How much it takes for the system to emit the alerts from the moment the anomalous transaction was produced.
- R1: Real time simulation. Measuring the mean response time from the start of the transaction of the detected anomalous scenario until the alert is emitted by the system.

Setting up the experiments:

- Scaling issue: - Fixed - scaled at the level of the μs

4.4.1 Small bank size & small transaction stream

- Scaling to 600s (10 minutes) → no losing of alerts (462).

My doubts on these experiments

- Results = Checks: same issue as with the alerts, a same check(tx1,tx2) on a card can not be performed until the tx2 reaches the system, which is the same simulated time on any system.

- For a small bank size, as the simulation tends to be closer to reality (lower scaling), then the system has lower overhead and the differences between the different variations will be almost negligible. Also the time needed to perform these simulations increases.
- However, on the contrary, when the scaling is high, then the simulation is farther from reality, making the system to be more loaded than in a real case scenario, reaching a point that the experiment will be almost the same as E2 (reading the transaction stream input directly without any real-time simulation).
- To have a "real-time" simulation in which differences between the systems can be observed we will need to simulate a big bank, which is able to provide a dense/big stream, which intuitively, even more if the stream is scaled, the simulation will be almost like reading the stream directly from the input like without any "real-time" simulation, like in the case of the E2 experiments.

Proposals:

- Do more variations on the stream size and bank database size for the E2 experiments.
  - Incrementing the database size, is expected to increase the overhead due to the greater number of cards and the latency to query the stable bank database.
  - Increasing the stream size...?
- On the E2 experiments measure and compare the response time for providing an alert (the time since the last transaction producing the alert happened until the time the system emits out that alert). **But focusing only on the alerts and not on all the checks, I think it is sufficient to compare only on the alerts.** Also measuring on the checks will introduce an overhead on the needed message passing from the filters to the sink to do all this gathering on these times, whereas with the alerts it is minimal since they are already sent to the Sink stage.

Some experiments that were tested:

- Scaled small bank data stream of 30 days to 10 minutes.
- Tested for these combinations:

# cores	# filters
1	1
4	40
16	2000

4.5 E2: Continuous delivery of results in a high loaded scenario

- Q2: Behavior of the different system variations on a high loaded transaction stream scenario.
- R2: Not real time simulation. Direct transaction stream input supply. Comparison of the continuous delivery of results of the different systems variations (diefficiency metrics).

4.5.1 Small bank size & small transaction stream

1-core

Amalia: Hay que explicar bien los resultados que se enseñan en cada grafica y lo que representan además de las conclusiones que permiten sacar.

Results for executions with 1-core:

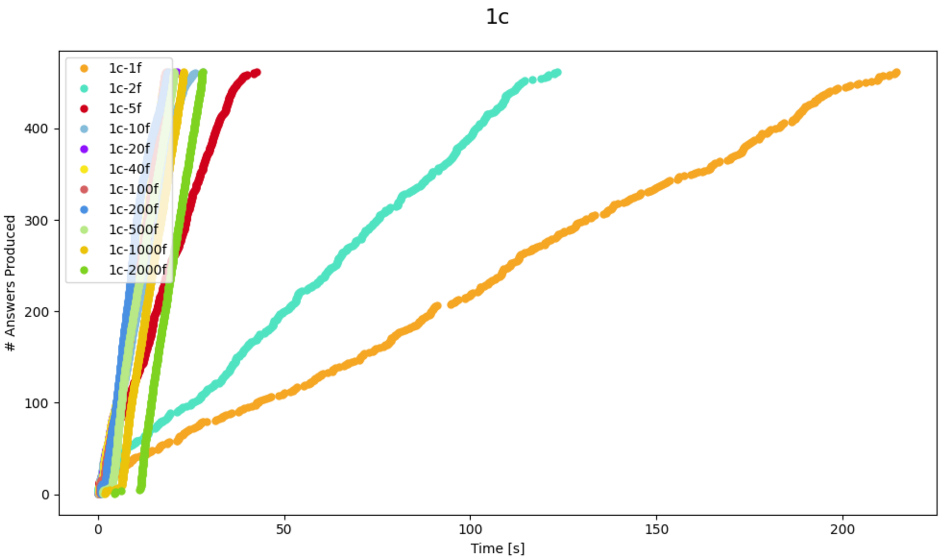


Figure 23: Alerts trace in time (s) - 1 core

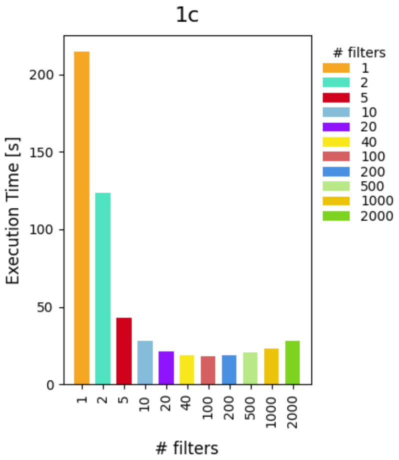


Figure 24: Execution time (s) - 1 core

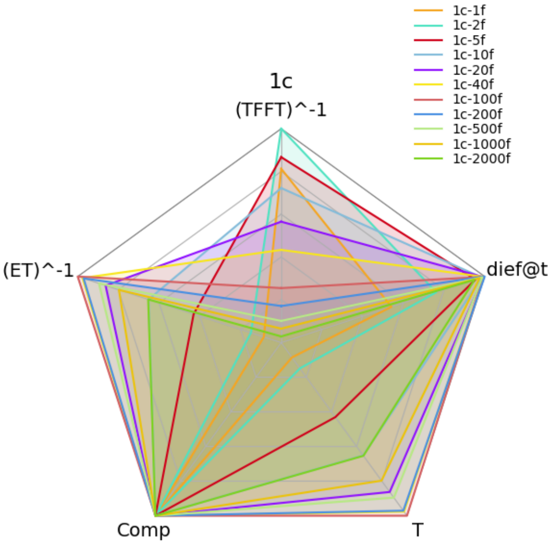


Figure 25: `dieft` radar - 1 core

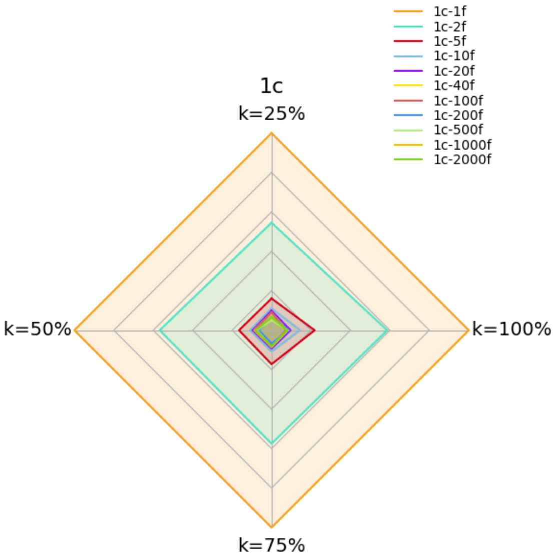


Figure 26: `diefk` radar - 1 core

TODO: Put table for not graphical results gathered in the dieffpy-out.txt file...

4-cores

Results for executions with 4-core:

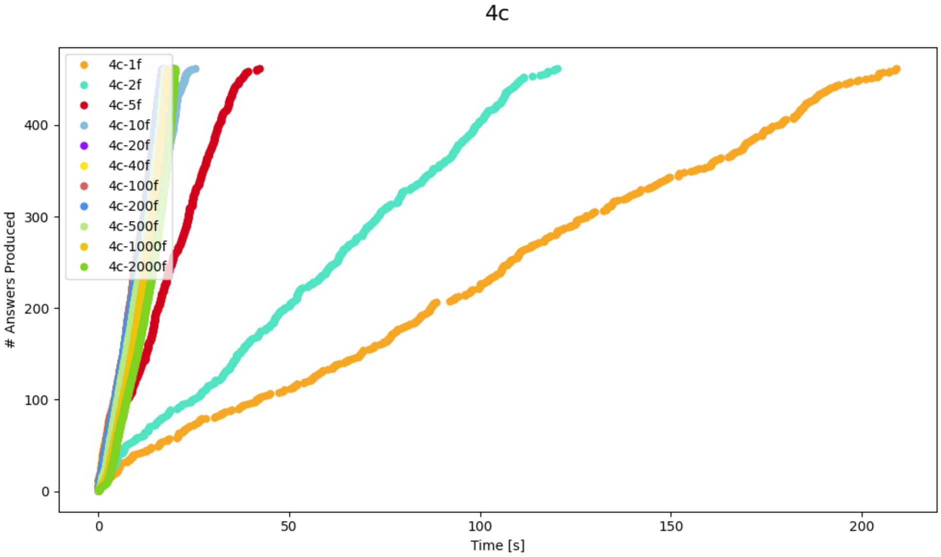


Figure 27: Alerts trace in time (s) - 4 core

8-cores

Results for executions with 8-core:

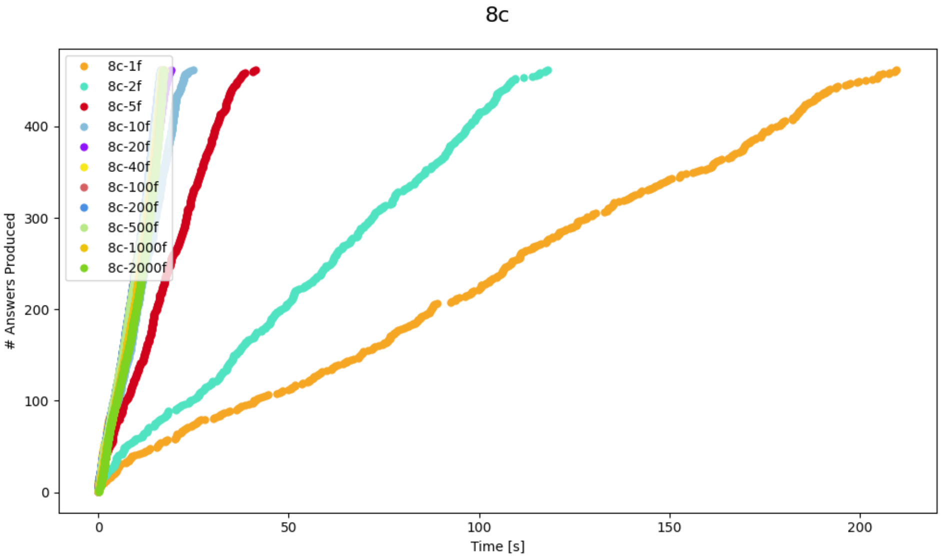


Figure 28: Alerts trace in time (s) - 8 core

16-cores

Results for executions with 16-core:

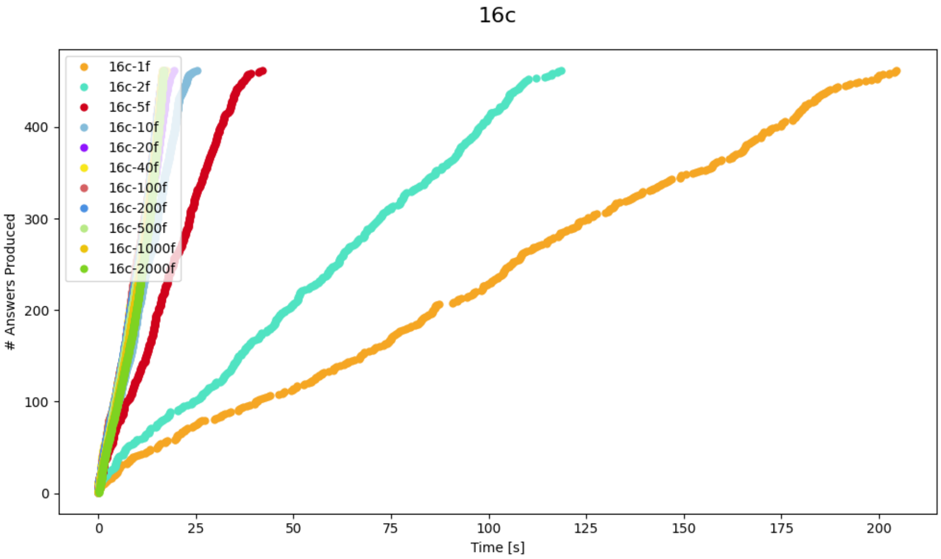
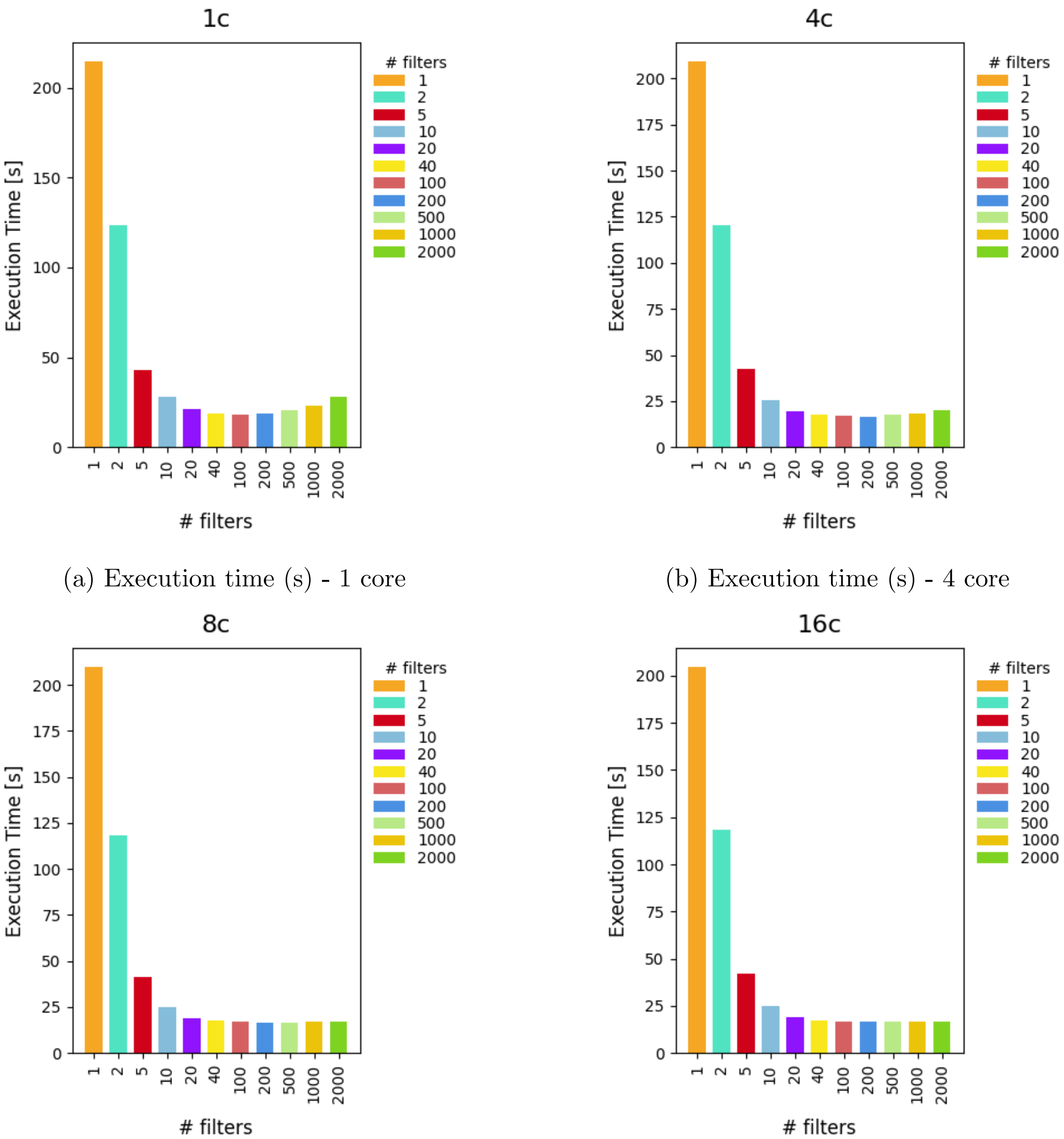


Figure 29: Alerts trace in time (s) - 16 core

NOTE: Almost no difference when variating the number of cores!!!!



(a) Execution time (s) - 1 core

(b) Execution time (s) - 4 core

(c) Execution time (s) - 8 core

(d) Execution time (s) - 16 core

Figure 30: Execution times for different core configurations.

Some observations:

- Differences between more or less cores visible specially for a fixed big number of filters (see plots of fixed number of filters) -¿ TODO
- More filters imply more overhead specially when number of cores is low.

Transaction stream - medium

- NUM_DAYS = 60

- `anomalous_ratio` = 0.02 (2%)

This setup gives us a transaction stream of

- `total_tx` = 80744
- `regular_tx` = 79005
- `anomalous_tx` = 1739

Transaction stream - big

- `NUM_DAYS` = 120
- `anomalous_ratio` = 0.02 (2%)

This setup gives us a transaction stream of

- `total_tx` = 160750
- `regular_tx` = 157756
- `anomalous_tx` = 2994

NOTE: only 5 runs each experiments instead of 10! - large execution time...

→ Issue: Connection timeout

In the cases:

- 1c: $\geq 200f$
- 2c: $\geq 1000f$

Possible fixes:

- (Optimization of the query... introducing indexing!)
- Increase the timeout
- Set a maximum connection pool size... Note that it may be needed for later experiments... Since we may not be able to open more than x connections/sessions in parallel with the Neo4j gdb. Then we have 2 options:
  - Limit the maximum number of filters based on this max limit on the number of parallel connections/sessions.
  - Do not limit the maximum number of filters, but instead create a pool of connections.

1. So far → Try to fix limiting the maximum timeout and limit the number of filters (with the idea that each filter has a permanent open session - instead of requesting a session from a pool of connections to a certain process manager of sessions every time it needs to query... - I think it is more clean and easy, and for the purposes of what we are doing to have a permanent open session per filter. So that:
   - Easier to manage (apparently no limit on the number of parallel sessions - so no problem)
   - We need to do a retry process - so that whenever the timeout exceeds it can try again without producing an error - and/or increase the timeout limit.
   - Finally note that this problem is going to appear whenever we have many open threads connected in a high loaded scenario tested on a low-resource variant of the system (with few/low number of cores). And that, for a real scenario this is not going to be the case since the system will be expected to be way less loaded.

→ Partial fix so far: increase the timeout of a transaction at the driver level to 1h: "config.MaxTransactionRetryTime = 1 * time.Hour"

Some details about our Neo4j VM:

- 4 cores and 20GB of RAM

- No limit on the number of parallel sessions. However it seems that by default there is a limit on the number of parallel transactions to 1000.

4.6 Update: Recording all the checks

4.7 How to measure MRT

To show the continuous delivery of results of the system, for testing purposes, instead of only recording the alerts (positive fraud checks) in the Sink, we are now going to be recording all the check results whether they are positive (alerts) or negative.
How we do this?
Measurement options:

- Measure the `time.End` of the check on the Sink. As with the alerts is the time it takes for the system to emit the result.

- Measure the `time.End` of the check on the Filter. It could be argued that the alerts (which are the unique results that in reality get out of the system) could be directly be sent from the Filters and not from the Sink, saving the time they need to travel to the Sink inside the system. Anyway they will need to travel to the Sink to be registered on the bank system but the alert to the user could be directly sent faster from the Filters.

So far:

- Measurement: done at the Sink.
- Register of the results at the Sink and sent all (positive and negative) through a unique channel from each Filter to the Sink (we reuse the `alert` channel).

Some experiments were done to inspect whether there was a significant difference on the measurement position. In particular, we measured the response time of each of the checks both in the Filter and in the Sink, to finally obtain the mean response metrics times of both approaches. In the table **??** we show the mean response time metric in ms measured at the Sink (`mrt-Sink`) and the difference of this measurement on average with respect to the measurement done in the Filter in ms (`mrt-difference`).
Note that each of this experiments were done running in not-real-time, with the small bank size, and the `30-0.02` stream. Each of the experiments were run 10 times... 16GB RAM...
As it can be observe the differences are negligible, and therefore for simplicity, and to maintain the "philosophy" of the dynamic pipeline, we decided to keep the measurement of the response times of the checks on the Sink stage.
Note that, measuring in the Filter would imply assuming that the alerts had to be sent from there, to be realistic to where we are measuring. Note that: measuring all the results and not only the alerts imply unnecesarily overloading the system, since only sending the alerts to the Sink should be enough for the purposes of our application. However, due to experimental purposes we are forced to send/measure all the checks, in order to be able to compare the continuous delivery of results of all the system configurations/variations.

# filters	# cores	mrt-Sink	mrt-difference
1	1	24899.103	0.071
1	4	24077.638	0.041
1	16	22302.060	0.016
40	1	8852.990	0.195
40	4	8012.537	0.065
40	16	8241.212	0.040
2000	1	13949.781	2.229
2000	4	10464.550	0.847
2000	16	7982.963	0.052

Table 7: Comparison of the response time measurement positions with different system configurations

4.7.1 Bank size: Medium

For these experiments, to generate the stream of tx, we needed to simplify this process in order to be able to generate a stream in a feasible amount of time. In particular we used the simplifed version of the `txGenerator.py`: `txGenerator-simplified.py`

$\rightarrow$ with a random ATM-subset instead of a closest to client ATM-subset. Also variation on the transaction distribution times.

- $|ATM| = 1000$
- $|Card| = 500000$

# cards per filter	# filters
500000	1
100000	5
50000	10
5000	100
2000	250
1000	500
500	1000
250	2000
100	5000
50	10000
10	50000

Transaction stream - small

- `NUM_DAYS` = 15
- `anomalous_ratio` = 0.03 (3%)

This setup gives us a transaction stream of

- `total_tx` = 4856573
- `regular_tx` = 4805920
- `anomalous_tx` = 50653

First run:

- 16GB RAM
- 16 cores
- Experiments for $|filters| \geq 100$

- x1 run each job

#cores	1f	5f	10f	100f	250f	500f	1000f	2000f	5000f	10000f	50000f
1											
2											
4											
8											
16	R	R	R	OK	OK	OK	OK	OK	OK	OK	outMem

TO TRY:

- 32 or 64 RAM
- 16 cores
- Experiments for $|filters| \geq 100$

5 How to run the experiments

- Run $> `launchAll-{1,2,4,...}c.sh <descriptions> <execTimes>` where we select the script to run based on the number of cores (1,2,4...) and maximum RAM with which to run the set of experiments. Indicate the directory of the description files of the experiments to run with `<descriptions>`, and the number of times to run each experiment with the `<execTimes>` parameter. Each description file of an experiment has to be in a csv format indicating `txFile,test,approach,maxFilterSize` where:

  - `txFile`: indicates the name of the input stream file.
  - `test`: label indicating the name of the test we perform (stream input and cores)
  - `approach`: label indicating the name of the approach we perform (cores and filters)
  - `maxFilterSize`: to set the maximum number of cards per filter. To set up the maximum number of filters for the tested system.

  An example of a csv experiment description file is shown in ??.

```
txFile,test,approach,maxFilterSize
../input/small/30-0.02.csv,30-0.02-1c,1c-4f,500
```

Listing 29: 30-0.02-1c-4f

- Run $> `summary-results.sh <directory> <TEST>`: to obtain the averaged results of the experiments run stored in the indicated output `<directory>` (the predefined output directory is the called `output` directory) and then `<TEST>` where we need to indicate the name of the performed test (like in the experiments description files).

5.1 Input reading by chunks

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

5.1.1 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

5.1.2 encoding/csv

5.1.3 Experiments over the different approaches

Approaches:

- `1-apache/arrow` direct reading of corresponding data type in the worker.
- `2apache/arrow` reading as string data type. Later conversion in main.
- `3-encoding/csv`: row by row reading and passing chunks of rows to main.

TODO: put a schema of the main/worker to show the different approaches better

- For the different approaches we tried with different sizes of files: 10^4, 10^5 and 10^6 number of rows (transactions).
- For each of the sizes we compared the time it took to read the full file to each of the variants, testing for different chunk sizes in terms of the number of rows: ranging from $10^0, 10^1, 10^2, ...$ up to the total number of rows of the file (maximum possible chunk size, all at once).
- Each of the experiments is done 20 times to obtain stable measurements.

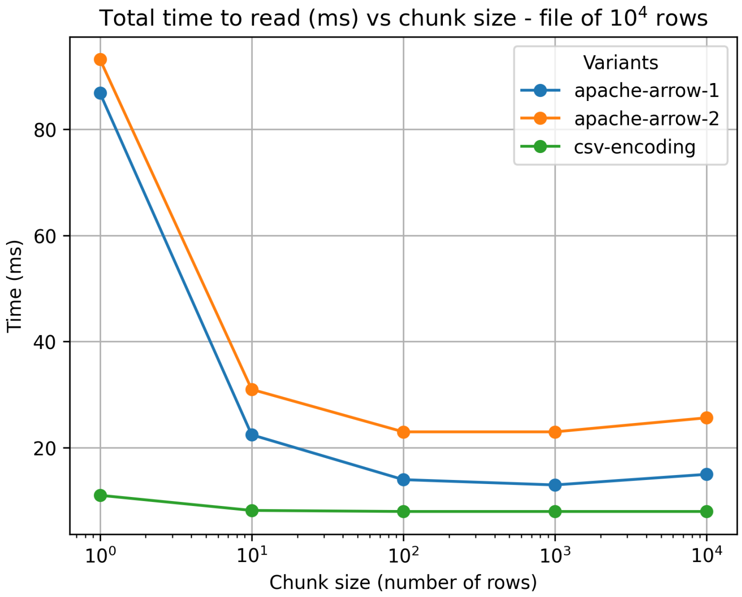


Figure 31: Comparison of the variants for file of 10^4 rows

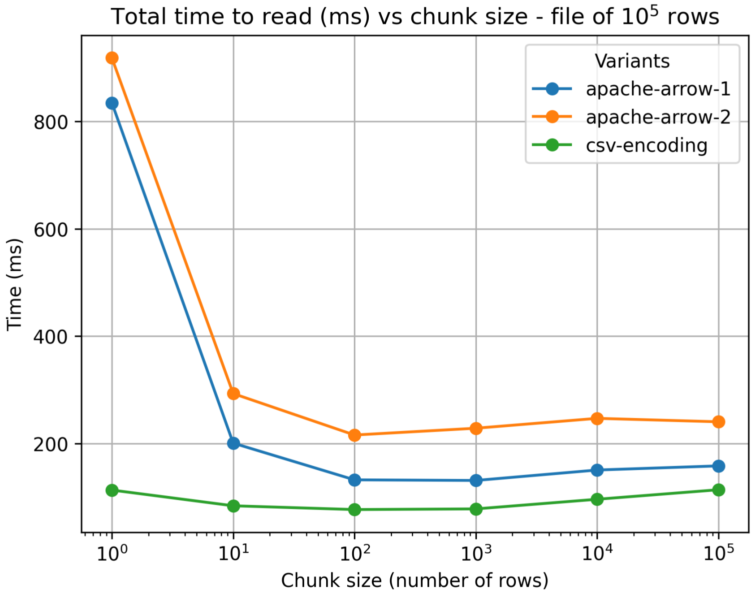


Figure 32: Comparison of the variants for file of 10^5 rows

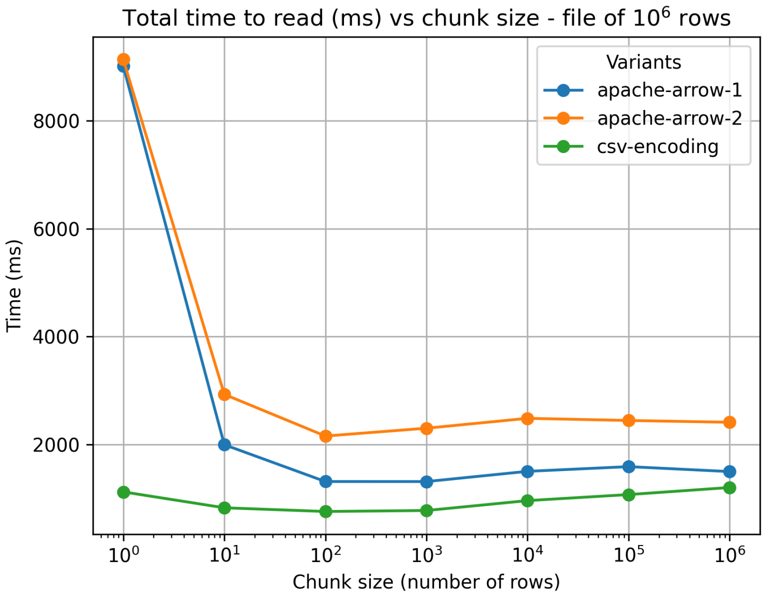


Figure 33: Comparison of the variants for file of 10^6 rows

Note that in all of the cases, the fastest approach is the one using the `csv/encoding` library. And, in addition, with chunk size of 10^2 rows.
Once we decided to use the approach using the `csv/encoding` library, we performed an additional experiment in order to see if it was actually worthy to do the *background* reading of the input with a worker goroutine. To see this:

- Compare the variant with worker and chunk size of 10^2 with the one without worker and therefore not reading by chunks.
- Comparison for different sizes of files: 10^4, 10^5 and 10^6 number of rows (transactions).
- Each of the experiments is done 20 times to obtain stable measurements.

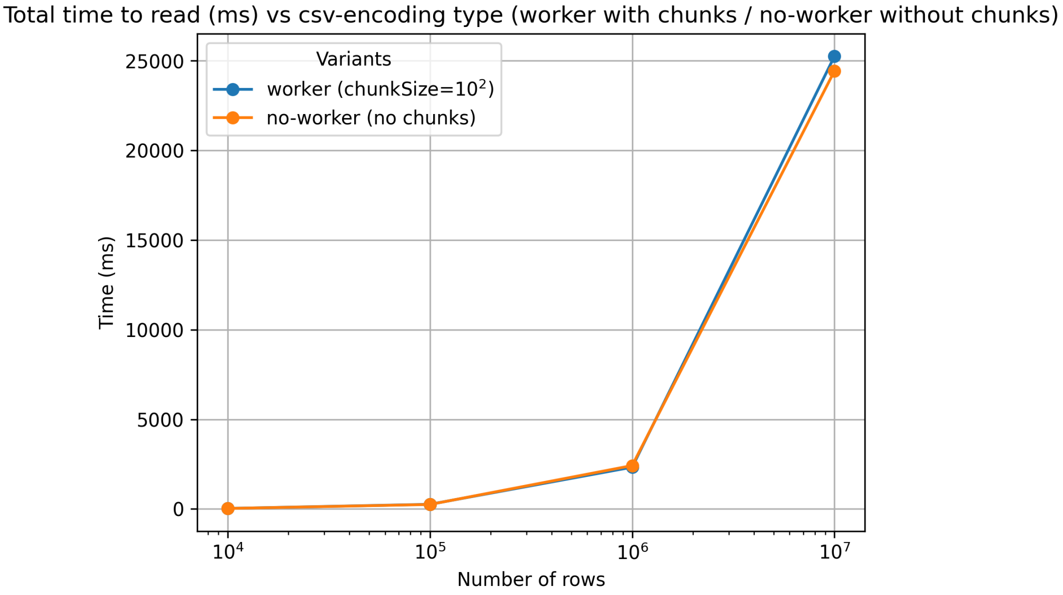


Figure 34: Comparison of `csv/encoding` variants up to 10^7 rows

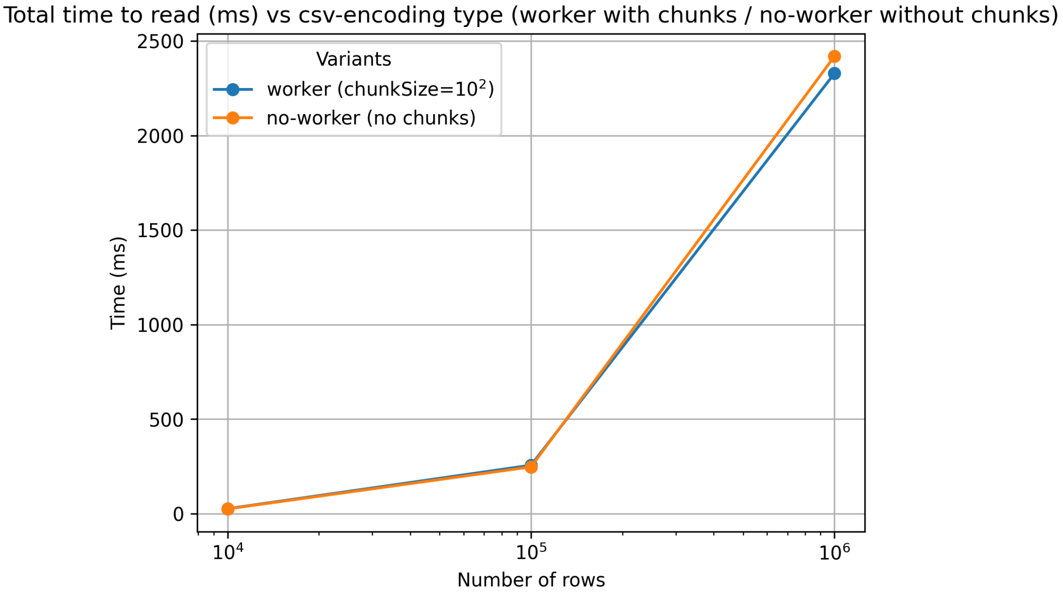


Figure 35: Comparison of `csv/encoding` variants up to 10^6 rows

- Differences insignificant
- Depend on the application
- Real-time simulation: worker version. To avoid possible bottleneck on the input reading. Instead the bottleneck be just the stopping to provide the input.

As it can be seen, the differences are insignificant, and the selection of each of the variants will depend on the application. For example, we suspect that the worker

version can be beneficial in the real time simulation, so that we do not make the reading be the bottleneck of the simulation, by having a background process reading input transactions from the stream files while the main process providing the input to the pipeline can be stopped doing the real time simulation.
TODO: Comparativa run same experiment NRT with worker VS without worker for the input providing - a single experiment example to show that is better with worker!
Some (other) references:

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

6 Conclusions

TODO:

- Lo que se hizo y cómo se hizo
- Resultados comentados. Pros y contras.
- Lo que faltó por hacer pero que sería interesante.
- Cómo se puede extender el trabajo a: mejores experimentos, más tipos de fraudes, otras aplicaciones...
- Windowing?