UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

CONTINUOUS QUERY ENGINE TO DETECT ANOMALOUS ELECTRONIC TRANSACTIONS PATTERNS USING BANK CARDS

FERNANDO MARTÍN CANFRÁN

Thesis supervisor
EDELMIRA PASARELLA SANCHEZ (Department of Computer Science)

Thesis co-supervisor
AMALIA DUCH BROWN (Department of Computer Science)

Degree
Master's Degree in Innovation and Research in Informatics (Advanced Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

30/01/2025

Abstract

Nowadays data are in motion, change continuously and are –potentially– unbounded implying data sources that are also in constant evolution. From the point of view of data persistence, this reality breaks the usual paradigm of dynamic although stable data sources. Besides, the number of applications to help critical decision making in real time is also rapidly increasing. These two scenarios raise the need of re-thinking both the data and the query models to fit these new requirements. So that, under these circumstances, it seems that a *continuously evolving data graph* is a suitable data model to use and therefore to study and analyze. Thus, in this work, we tackled the problem of querying continuously evolving data graphs in a specific context: the context of ATM[1] transactions, in particular anomalous ones. Under this context, evaluating continuous queries corresponds to recognizing patterns –usually associated with anomalous behaviors– in the *volatile* (evolving) subgraph of ATM transactions. To do so, we propose an evaluation process based on the so called *dynamic pipeline computational model*, a stream processing technique that facilitates the emission of alerts as soon as anomalous patterns are identified. Stream based bank applications that monitor ATM transactions are direct beneficiaries of our proposal since they can continuously query data graphs to get "fresh" data as they are produced, avoiding the computational overhead of having to discard non-valid data, as current systems work.

Keywords— continuous query evaluation, property graphs, dynamic pipeline approach, stream processing, ATM transactions

[1]Automated teller machine

Acknowledgements

I would like to express my gratitude to my teachers Edelmira Pasarella and Amalia Duch, whose guidance and support have been invaluable throughout this work. They have been always patient and have offered me great opportunities for my professional career that I will never forget. Thank you for everything.

In addition, I want to express my deep gratitude to Daniel Benedí, who has been a helping hand whenever I needed support throughout this long journey. Also to all the colleagues and friends from the master program.

I am also grateful to all the friends I have get to know during this chapter of my life in Barcelona. Without you, these two years would not have been the same. And, of course, to my friends from my hometown, Zaragoza, who have had the ability to stay by my side during these years, even if I have sometimes not spend much time with you.

Finally, and most importantly, to my family. Especially to my parents Jesús and Maria del Carmen, my sister María and to my grandmother Teresa, always in my memories. Thank you for everything. You are the best thing of my life.

Contents

1 Introduction **5**

2 Related Work **9**

3 Preliminaries **10**
3.1 Graph Databases 10
3.2 Dynamic Pipeline Approach 12
3.3 Metrics 13

4 Proposal **15**
4.1 Modeling and Implementing the Continuous Query Engine 15
4.2 Defining Anomalous Patterns of Transactions 16
4.3 Data Model 20
4.4 The Query Model 27

5 Continuous Query Engine - $\mathsf{DP}_{\mathsf{ATM}}$ **31**

6 Experiments **46**
6.1 Design of Experiments 46
6.1.1 E0: Evaluation in a Real-World Stress Scenario 47
6.1.2 E1: Evaluation in a High-Load Stress Scenario 48
6.2 Experimental Settings 49
6.2.1 Hardware 49
6.2.2 Software 50
6.3 Datasets 50
6.3.1 Synthetic Bank Database Creation 51
6.3.2 Synthetic Transaction Stream Generation 57
6.4 Considered Datasets 65
6.4.1 Stream Configurations 66
6.5 Stream Ingestion 67

7 Analysis of Experimental Results **71**

8 Conclusions **84**

1 Introduction

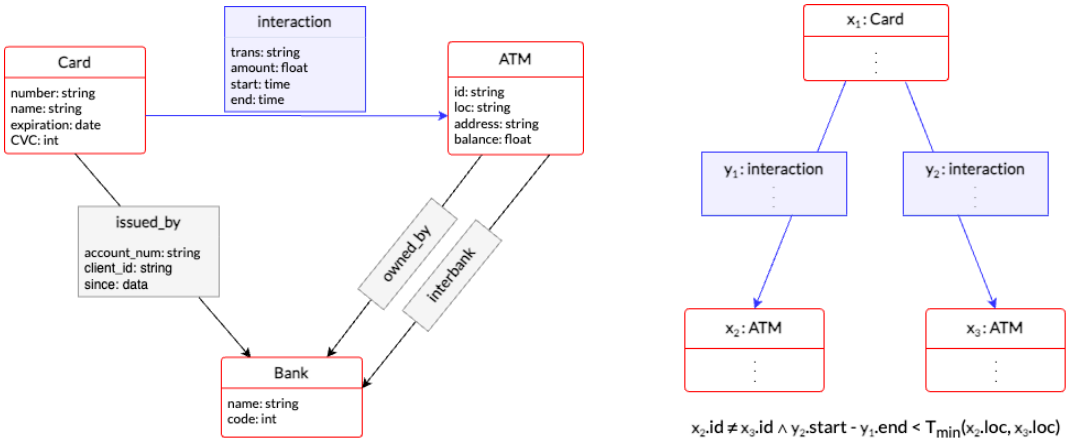
As an example of critical decision making scenario in real time, we consider the problem of detecting anomalous usage patterns in bank card transactions. In particular, suppose a card is used to withdraw money in two different locations in an elapsed time interval that makes it impossible for the cardholder to physically move from one location to the other; let us suppose, for instance, that one ATM is in Barcelona and the other in Madrid and two transactions are made with the same card in less than an hour, one at each ATM. Note that we are not concerned here with the typical physical sabotage of ATMs to steal/clone data, but instead we are concerned with possible criminal situations beyond sabotage, such as cloned bank cards. While using a cloned bank card chances are that suspicious patterns like the one described above arise. This is exactly what our proposal aims to detect in order to prevent a crime from being carried out, thereby protecting the interests of both, the bank and the cardholder.

Although from a classical point of view databases consider data as persistent [1], nowadays this perspective is changing [10, 54] as data is in motion, continuously changing and (possibly) unlimited. This is the case of ATM transactions. Indeed, transactions occur continuously and are not (necessarily) bounded. In this setting, the fact that transactions should be somehow represented in banking databases makes us reconsider the meaning of their persistence in banking systems raising the following two questions: (i) What is the appropriate data model to deal with this kind of transactions? and (ii) What is the appropriate query model?

Regarding the data model, the new nature of data requires a *de facto* new database paradigm -*continuously evolving databases*- where data can be both *stable* and *volatile*. Even though evolving databases can be implemented according to any approach, graph databases seem especially well suited here [6, 23]. 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 in data streams during a set time interval. Once this time interval is over, the relations are no longer valid so that there is no need to store them in the (stable) graph database. Volatile relations induce subgraphs that exist only while the relations are still valid. In this work we tackled the problem of evaluating continuous queries corresponding to *anomalous patterns* of ATM transactions against a continuously evolving graph database representing a bank database.

Without loss of generality we use *Property Graphs* (PG) [7, 5] as the basic reference graph 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.

Concerning the query model, fixed queries evaluated over data streams are known as *continuous queries* [11, 55]. Thus, instead of classical query evaluation processes we envision *incremental/progressive* query evaluation processes. A query on a PG database can be seen as a PG graph pattern with constraints over some of its properties. Evaluating such



(a) Part of a schema of a PG (b) Pattern of anomalous transactions

Figure 1: Part of a PG schema specifying volatile (interaction edges) and stable (issued_by, owned_by, interbank edges) relations in an evolving ATM Network and a continuous query pattern.

a query consists on identifying if there is a subgraph of the database that matches the given pattern and satisfies its constraints. Figure 1b depicts the pattern corresponding to an anomalous situation in the volatile (PG) subgraph of the considered database. This is, a constrained graph pattern corresponding to a continuous query. An anomalous pattern of ATM transactions must be identified in the volatile (PG) subgraph of the considered database.

The problem of progressively identifying and enumerating bitriangles (i.e. a specific graph pattern) in bipartite evolving graphs using the *Dynamic Pipeline Approach* (DPA)[36] have been successfully solved by Royo-Sales [42]. Let us observe that the problem of evaluating continuous queries over *continuously evolving PGs* belongs to the same family of problems and hence we propose to address it using the same stream processing computational model. Figure 2 illustrates the model associated to the problem described at the beginning of this section. In addition to identify the query pattern, the constraint satisfac-

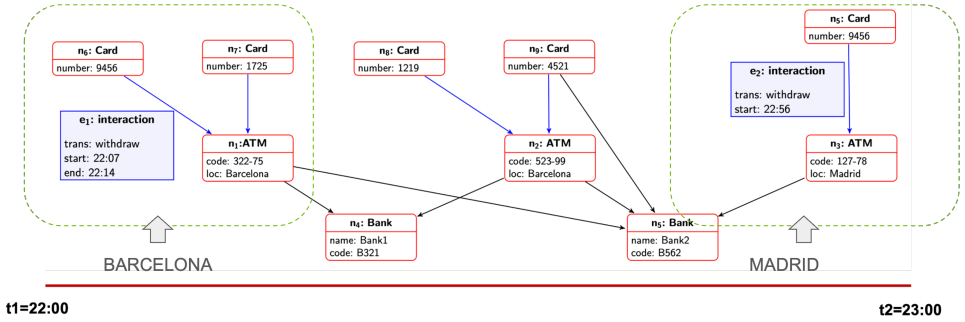


Figure 2: Example of the occurrence of anomalous ATM transactions in (a part of) a continuously evolving PG over a time interval: the card 9456 is used twice at ATMs in different cities, within one hour. However, to get from one of the cities to the other and vice versa requires more than one hour using any means of transport. This example could represent a possible crime pattern after a case of *skimming* and *cloning*.

tion over properties must be checked also. The evaluation process, done by a *Continuous Query Engine* (CQE) is based on the dynamic pipeline computational model [36] and emits answers (alarms) as soon as anomalous patterns are identified. Moreover, in a real application, when required -as for further legal or auditing purposes- timestamped occurrences of volatile relations can be kept in a log file. Figure 3 depicts a basic architecture of a CQE.

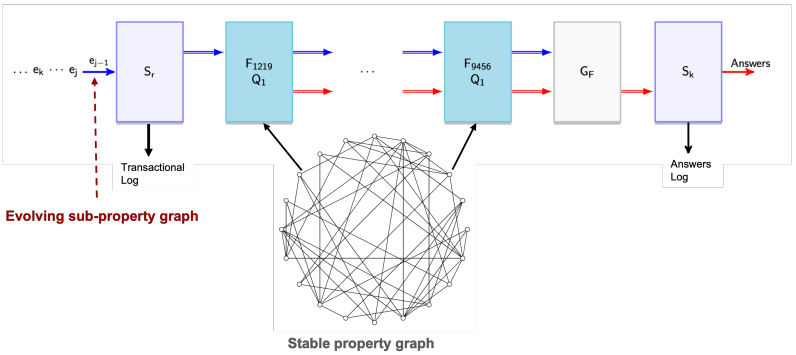


Figure 3: Preliminary continuous query engine architecture for detecting anomalous ATM transactions based on the dynamic pipeline computational model. Considering the schema given in Figure 1a, in this directed (multi) graph presentation of the $\mathsf{DP}_{\mathsf{ATM}}$, the arriving input data is a stream $\langle \dots \mathsf{e_k} \dots \mathsf{e_j}\, \mathsf{e_{j-1}} \rangle$ corresponding to interactions (volatile relations). Boxes (vertices) represent *stateful* processes called *stages* and internal arrows in the pipeline represent channels. Blue channels carry interaction edges and red channels carry detected anomalies (answers). $\mathsf{S_r}$ and $\mathsf{S_k}$ correspond to the Source and Sink stages which receive input data and results, respectively. Filter stages, $\mathsf{F_N}$, are parameterized with the value of the property number (N) of Card vertices. The Generator stage, $\mathsf{G_F}$, is in charge of spawning new filters, when required. The stable PG is a standard bank database (i.e. without volatile relations). Transactional log and Answers log keep input interactions and answers, respectively.

In order to test the practical suitability of our proposal, we have implemented it in the Go programming language since this language easily provides connection to existing implementations of Neo4j graph databases such as the one we use in this work, in addition to providing the necessary tools to exploit the parallel and distributed properties of the DPA with which we represent our proposal. Additionally, in order to experiment with the performance of our model, we provide a program capable of generating synthetic datasets useful to simulate bank graph databases including several parameters such as the users, their card holders, their ATMs geographically distributed, the stream of transactions, etc.[14] We assessed our model with two different bank sizes considering standard metrics as well as specific ones [2] for measuring continuous behavior and we could observe that (compared to a sequentially implemented system) the proposal using the DPA performs clearly better, providing almost immediate responses to the tested queries. Additionally, as opposed to artificial intelligence approaches [3] to predict frauds, ours provides 100% of effectiveness detecting fraud patterns. This is, there is no place to consider false alarms (false positive recognition of pattern) or ignore alarms due to non-recognition of patterns (false negative) since, in presence of the studied fraud pattern, the CQE will recognize it. Partial results of this work were reported and presented in the Alberto Mendelzon Workshop 2024 [26]. We can summarize the contributions of this work as follows.

Contributions.

- A general technique for addressing the problem of continuous query evaluation against an evolving graph database by decomposing the data graph into *volatile* and *stable* well defined subgraphs, using a stream processing approach. Among the advantages of using the dynamic pipeline computational model are its parallel/concurrent nature and its suitability for developing real-time systems that emit results as they are computed, in a progressive way.

- A characterization of some possible fraud graph patterns. Even non-exhaustive, to our knowledge, it is a first specification of what must be consider a fraud pattern in terms of (continuous) graph databases. This characterization is useful not only when considering ATM transactions but, in general, for any bank cards online transactions.

- A continuous query engine to detect abnormal or suspicious ATM transactions. To our knowledge, most of the work addressing this topic provide a delayed detection based on predictions given by ML systems. Also, it is frequent the classical treatment of the problem by consulting log files because of the complaint of customers when detecting by themselves some weird movement in their accounts. This involves annoying processes for customers in order to have their money back. The idea is that, in presence of some weird finding in an ATM transaction, banks have a tool able to either ask card holders for authorizations or to take any action preventing other fraud at real-time.

- Given the sensitive nature of banking data and transactions, there are no repositories that offer this type of datasets for empirical studies. In this work, we have created an open synthetic repository for this purpose.

The rest of this work is organized as follows. In Section 2, we describe the related existing work while, in Section 3, we give the required preliminaries to follow the whole work regarding graph databases, graph fraud patterns, the dynamic pipeline paradigm and the metrics used to evaluate the features of our proposal. In Section 4, we describe the details of our proposal by defining anomalous patterns of transactions, how to model and implement the continuous query engine, the used data model and query model as well as the architecture of the continuous query engine. Section 5 is devoted to the continuous query engine, the $\mathsf{DP}_{\mathsf{ATM}}$. The experiments are described in Section 6 where the different evaluation scenarios, the experimental settings, the datasets and the stream configurations are considered. The experimental results are analyzed in Section 7 and in Section 8 we give some conclusions and proposals for further work.

2 Related Work

Recently, in Rost et al. [41] authors formalize an extension to the query language Cypher that allows for evaluating continuous queries over property graph streams according to a specified frequency, during a defined time interval. The main differences of their approach with our proposal are: first, the data model because they merge incoming PGs to the persistent database and, second, we evaluate continuous queries as new edges are added to the volatile part of the property graph (i.e. as the PG evolves) by processing (identifying) patterns (subgraphs). To our knowledge there is no other work approaching the problem of modeling and implementing a continuous query engine for detecting anomalous ATM transactions. In effect, according to the survey of Rahman et al. [37] most of the ATM fraud detection and prevention mechanisms are based on analyzing the spending behavior of cardholders in order to detect suspicious behavior. Furthermore, these authors claim that all these mechanisms are based on artificial intelligence, data mining, neural networks, Bayesian networks, artificial immunology systems, support vector machines, decision trees, machine learning, etc. The most important criteria to be consider for assessing these kind of methods are their accuracy, speed and cost. Accuracy refers to the capacity of effectively detecting a fraud, i.e. is a metric that measures how often a machine learning model correctly predicts the outcome. It can be computed by dividing the number of correct predictions by the total number of predictions. In Rahman's survey [37], the relevant result after comparing different methods is that the accuracy is low in most of them. In addition, these methods need training processes and hence, big volume of data. In the same line is the work of Ahmed et al. [3] in which, again, the importance of detecting frauds in financial domains and some methods based on artificial intelligence techniques are analyzed. To be concrete, authors study methods based on clustering anomaly detection techniques. As we do, authors complain about the lack of real world data, one of the point we tackle in this work. In [22] authors present a system to block fraudulently issued transactions based on a big data clustering method to timely identify abnormal transactions. They claim that by clustering credit card data and its transactions, it is possible to identify frequent and expensive purchases, and then to investigate the possibility of a crime to discover specific cases. In the conclusions of this work the need of identifying fraud patterns in the use of bank cards is highlighted. However, it is not proposed a specific way to model and deal with these patterns. Another work [19] provides a fraud recognition models based on debit card transaction dataset from Indonesian bank. The starting point of this research is that "fraudulent transaction contains 'anomaly' from the pattern of non-fraudulent transactions". Authors use classification models to detect anomalous patterns, obtaining an accuracy around 0.7 in both cases. Although there is a lot of work tackling the problem of detecting anomalous ATM transactions, most of them based on artificial intelligence techniques, there is not a clear characterization of the problem and what to solve it means or implies. As said before, we focus on providing an explicit characterization of what anomalous transaction patterns are and a method based on graph recognition and stream processing computational model to detect them, with 100% of accuracy.

3 Preliminaries

For the interested reader, in this section we introduce the basic definitions and notions that are used in the forthcoming sections of this work, we also introduce the basic notation that we are going to use.

3.1 Graph Databases

Graph Data Model: Property Graph

A data[base] model is a collection of conceptual tools used to model representations of real-world entities and the relationships among them [6]. A graph database model is a data[base] model for the understanding and management of graph data [23]. There are three dominant graph data models: the property graph, Resource Description Framework (RDF) triples, and hypergraphs. A commonly employed graph data model in practice is the property graph data model [39, 7]. This will be our selected graph database model.

Informally, a property graph (PG) is a directed labeled multigraph with the special characteristic that each node or edge could maintain a set (possibly empty) of property-value pairs. 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. More formally, we provide the same definition as the one in [5]:

Definition 1 *A property graph is a tuple $G = (N, E, \rho, \lambda, \sigma)$ where:*

1. *N is a finite set of nodes (also called vertices);*
2. *E is a finite set of edges such that E has no elements in common with N;*
3. *$\rho : E \to (N \times N)$ is a total function that associates each edge in E with a pair of nodes in N (i.e., ρ is the usual incidence function in graph theory);*
4. *$\lambda : (N \cup E) \to SET^{+}(\boldsymbol{L})$ is a partial function that associates a node/edge with a set of labels from L (i.e., λ is a labeling function for nodes and edges);*
5. *$\sigma : (N \cup E) \times \boldsymbol{P} \to SET^{+}(V)$ is a partial function that associates nodes/edges with properties, and for each property it assigns a set of values from $\boldsymbol{V}$.*

Where $\boldsymbol{L}$ is an infinite set of labels (for nodes and edges), $\boldsymbol{P}$ is an infinite set of property names, $\boldsymbol{V}$ is an infinite set of atomic values, and $\boldsymbol{T}$ is a finite set of datatypes (e.g., integer). Given a set X, we assume that $SET^{+}(X)$ is the set of all finite subsets of X, excluding the empty set. Given a value $v \in \boldsymbol{V}$, the function type(v) returns the data type of v. The values in $\boldsymbol{V}$ will be distinguished as quoted strings.

Given two nodes n_1, $n_2 \in N$ and an edge $e \in E$, such that $\rho(e) = (n_1, n_2)$, we will say that n_1 and n_2 are the "source node" and the "target node" of e respectively. Additionally, given a property $(o, p) \in (N \cup E) \times \mathbf{P}$ and the assignment $\sigma(o, p) = \{v_1, ..., v_n\}$, we will use $(o, p) = v_i$ with $1 \leq i \leq n$ as a shorthand representation for a single property where o is the "property owner", p is the "property name" and v is the "property value". Note that their definition supports multiple labels for nodes and edges, and multiple values for the same property.

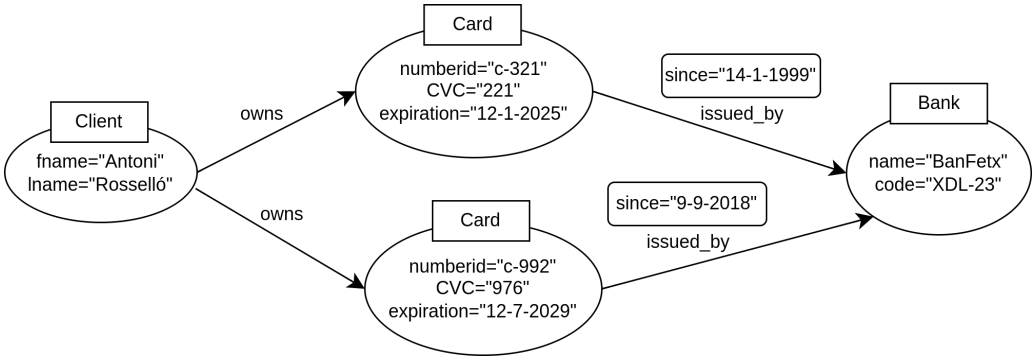


Figure 4: Property graph example representing bank database information

As an example we provide Figure 4, where we show a property graph representing bank data information. With respect to the formal definition given we have:

- $N = \{n_1, n_2, n_3, n_4\}$
- $E = \{e_1, e_2, e_3, e_4\}$
- $\lambda(n_1) = \{\text{Client}\}$, $(n_1, \text{fname}) =$"Antoni", $(n_1, \text{lname}) =$"Rosselló"
- $\lambda(n_2) = \{\text{Card}\}$, $(n_2, \text{numberid}) =$"c-321", $(n_2, \text{CVC}) =$"221", $(n_2, \text{expiration}) =$"12-1-2025"
- $\lambda(n_3) = \{\text{Card}\}$, $(n_3, \text{numberid}) =$"c-992", $(n_3, \text{CVC}) =$"976", $(n_3, \text{expiration}) =$"12-7-2029"
- $\lambda(n_4) = \{\text{Bank}\}$, $(n_4, \text{name}) =$"BanFetx", $(n_4, \text{code}) =$"XDL-23"
- $\rho(e_1) = (n_1, n_2)$, $\lambda(e_1) = \{\text{owns}\}$
- $\rho(e_2) = (n_1, n_3)$, $\lambda(e_2) = \{\text{owns}\}$
- $\rho(e_3) = (n_2, n_4)$, $\lambda(e_3) = \{\text{issued\_by}\}$, $(e_3, \text{since}) =$"14-1-1999"
- $\rho(e_4) = (n_3, n_4)$, $\lambda(e_4) = \{\text{issued\_by}\}$, $(e_4, \text{since}) =$"9-9-2018"

Graph Database System: Neo4j

As defined in [39], a graph database management system (henceforth, a graph database) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind.

One of the most popular graph databases today is `Neo4j` [23]. `Neo4j` is a disk-based transactional graph database. It is an open source project available in a GPLv3 Community edition, with Advanced and Enterprise editions available under both the AGPLv3 as well as a commercial license. `Neo4j` is best graph database for enterprise deployment. It scales to billions of nodes and relationships in a network. It works based on the property graph model.

`Cypher` [30] is a declarative query language that allows for expressive and efficient querying, updating and administering of a `Neo4j` graph database.

In our case we will use `Neo4j` as graph database system, and we will interact with it using the `Cypher` query language.

3.2 Dynamic Pipeline Approach

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 Approach (DPA) [36]. The DPA 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 approach 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.

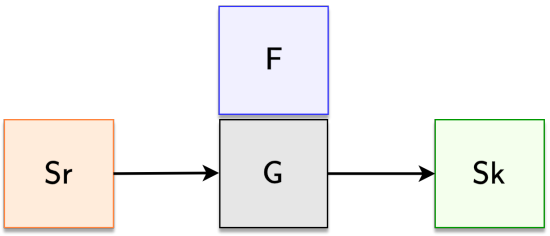


Figure 5: Initial configuration of a Dynamic Pipeline. An initial DP consists of three stages: *Source* (Sr), *Generator* (G) and *Sink* (Sk). Above the *Generator* (G) the *Filter* (F) parameter. The stages are connected through its channels, represented with the black right arrows.

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 5 represents the initial configuration of a DP and Figure 6 depicts the stages of the DP after a possible evolution, where the *Generator* has created two *Filter* stage instances.

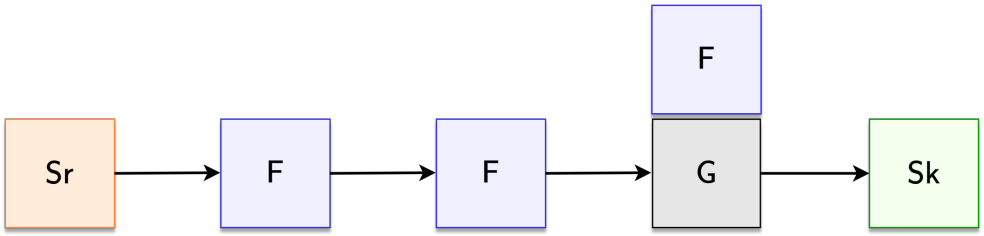


Figure 6: Evolution of a DP. After the creation of two *Filter* F stage instances of the *Filter* parameter (above the G stage) by the *Generator* G stage.

The DPA has been used to model many different problems. In [42] they successfully solved the problem of progressively identify and enumerate bitriangles (i.e. a specific graph pattern) in bipartite evolving graphs. In [24] DPA is used to model the problem of multidimensional range queries, that is, selection queries on objects in a k-dimensional space. Finally, in [13] an algorithm to solve the problem of computing and maintaining in practice the minimum spanning tree of dynamic graphs using the DPA is proposed and tested experimentally.

In our case, we envision here the architecture of a continuous query engine, the $\mathsf{DP}_{\mathsf{ATM}}$, to detect anomalous ATM transactions under the DPA, where the continuous input data stream is the stream of the bank ATM transactions, and the activity of a Card is tracked using the *Filter* stages of the DP (see Figure 6).

3.3 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 [2]. 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 a transaction interaction arrives to the system until the result of its respective fraud pattern check is produced. The `MRT` is the average response time metric for all the results emitted by the system. In a real-time system like ours we are interested in low values of these metrics.
- **Execution Time (`ET`)**
  It measures the total time (in seconds) that it takes for the system to consume/process a full input stream. The lower, the better.

- **Throughput (`T`)**
  It measures the number of results emitted per time unit. It is calculated as number of results divided by `ET`. The higher the throughput the better in relation to the performance of our system.

- **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`. The higher the value of the interactions per second the better in relation to the performance of our system.

- **Time to produce the First Tuple (`TFFT`)**
  It is the time required by the system to produce/emit the first result. The lower the value of this metric, the better. Meaning that the system is able to start up fast.

- **`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, they 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 [2]. 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 [44], which calculates them from the generated answer 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 and modified the tool for our specific needs. This was done in order to visualize other metrics, especially the `MRT`, but also others like the throughput `T`, `interactions/s` or the `TFFT`.

4 Proposal

Defining and implementing a continuous query engine requires to address many different problems, each of different nature. In addition, the proof of concept we intend to provide has itself its own complications. We address all of them in this section.

4.1 Modeling and Implementing the Continuous Query Engine

To define a proper architecture for a continuous query engine is one of the most challenging activities of our work. Among other tasks, this comprises: to define a graph-based query language expressive enough that allows capturing the different patterns representing anomalous queries; to establish the algorithms for identifying the patterns associated to anomalous queries; to choose and manage the right windowing approach and other features related to distributed query-evaluation; to deal with the evaluation of many continuous queries simultaneously; to evaluate the suitability of the implementation language, tools and the proper system configuration. Figure 3 depicts a preliminary architecture for a continuous query engine for detecting anomalous ATM transactions, $\mathsf{DP}_{\mathsf{ATM}}$. Figure 7 depicts an abstraction of the architecture of the Continuous Query Engine that we propose and show how this system can be used to prevent ATM transactions frauds combining it with a double check mechanism.

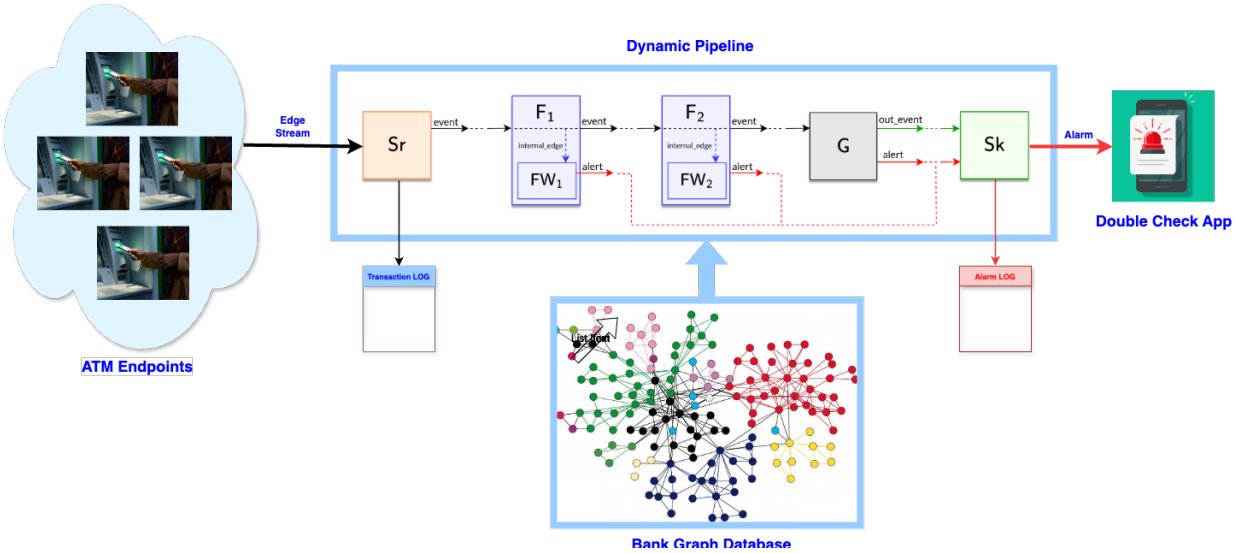


Figure 7: Abstraction of the Continuous Query Engine for detecting anomalous ATM transactions based on the dynamic pipeline computational model. In this abstract architecture of the Continuous Query all its components as well as its expected usage are shown.

As we said, we propose a solution that follows the Dynamic Computational Approach [36]. Briefly speaking, in this approach, *stages* are processes that execute tasks concurrently/in-parallel. The multiset underlying the input data stream is partitioned [12] and distributed along filters according to a grouping relationship, usually based on filters' parameters. Each filter applies the same function to its block of data (stored as its state). Accordingly, the $\mathsf{DP}_{\mathsf{ATM}}$ algorithm is specified as follows: During a pre-defined time interval window,

when an interaction e (together its properties' values) arrives to the $\mathsf{DP}_{\mathsf{ATM}}$, the stage S_r register it into a standard transactional log file. Then, S_r passes e to the next stage. If there exists a filter parameterized with the value of the property number of the Card vertex that is incident to e, this filter keeps e in its state. In this way, filters' states store subgraphs induced by the edges in the volatile subgraph. Notice that these sets of edges in each filter correspond to blocks of the (multiset) input data stream. Otherwise, the filter passes e to the next stage. The task/function of each filter is to decide if there is a match with (some of) the continuous query pattern(s) evaluated by the engine $\mathsf{DP}_{\mathsf{ATM}}$ by means of the graph that it stores and the information retrieved from the stable PG to identify patterns and solve constraints. This is, indeed, the way to evaluate continuous queries. In case of matching a pattern, filters emit an alert reporting the finding. Hence, answers are the detected anomalies and they are emitted as they are obtained in filters. When answers arrive to S_k, this stage post-processes and output them. In addition, S_k 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 and thus, a new filter parameterized with this new value is spawned. 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.

4.2 Defining Anomalous Patterns of Transactions

It is not trivial to establish what is and in which circumstances a transaction can be considered anomalous. Based on a work that have addressed this characterization [25] 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. Figure 1b depicts an example characterizing a possible card cloning, among many other possibilities. For instance, using a (stolen) card many times over a period of time at different ATMs to withdraw small amounts. In this latter case, there will arrive to the evolving PG many volatile (interaction) edges having the same card vertex and different ATM vertices. There could also be patterns related with frequent/very high expenses; transactions located in an ATM out of the threshold distance of the usual/registered address of the card holder and so on. Moreover, definition of patterns can be beyond ATM transactions by considering Point-Of-Sale (POS) or online card transactions.

I. Card cloning characterization

II. Lost-and-stolen card characterization

III. Other possible fraud scenarios

I - Card Cloning Characterization

Card cloning can be defined as "a kind 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" [47].

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.

The method consists on detecting abnormal card-ATM activity of the same card at different ATMs taking place within an unfeasible time distance difference. That is, when a transaction is made at an ATM, and after that, another transaction is initiated with the same card at a different ATM, such that the distance between the two is impossible to be covered within the time between the transactions. The detection of this anomalous scenario is represented on the PG graph pattern of Figure 8.

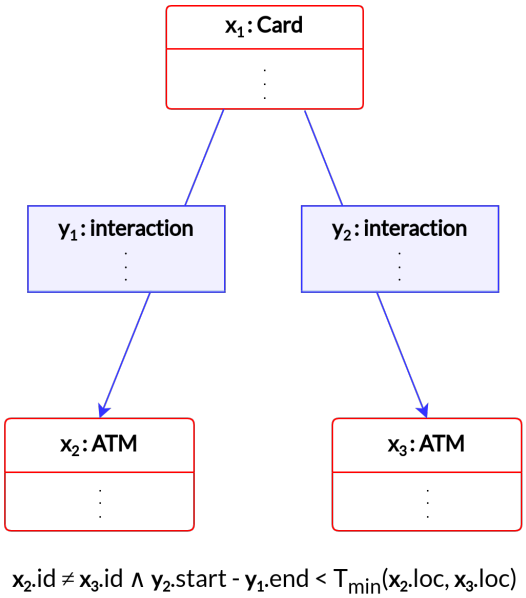


Figure 8: Property graph pattern - Card cloning characterization

The pattern consists on a `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 $x_2.location$ represents the location coordinates pair of the x_2 ATM: $x_2.location = (x_2.loc\_latitude, x_2.loc\_longitude)$. Same for the `ATM` x_3.

An example of this kind of anomalous card-ATM interaction, could be one as represented on Figure 9, 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 interval.

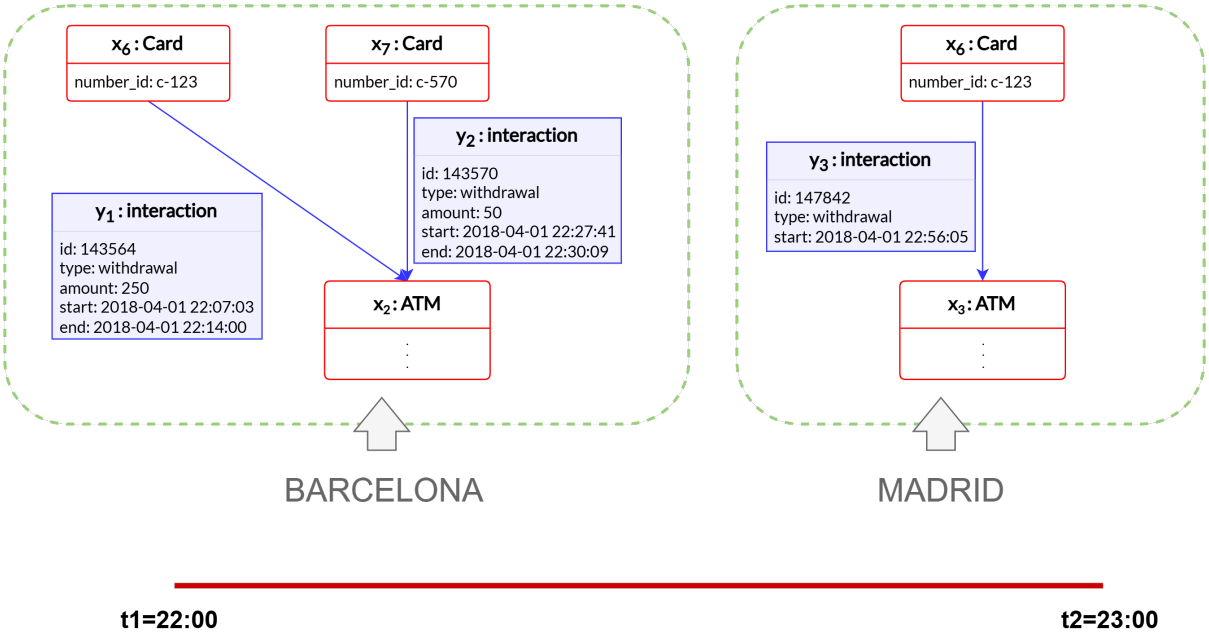


Figure 9: Card cloning characterization - an example

II - Lost-and-Stolen Card Characterization

"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" [4].

A possible way that we propose to detect this kind of fraud 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 become aware of the loss of the card and asks the bank to freeze it. The detection of this kind of fraud scenario is modeled with a PG graph pattern as the one represented in Figure 10.

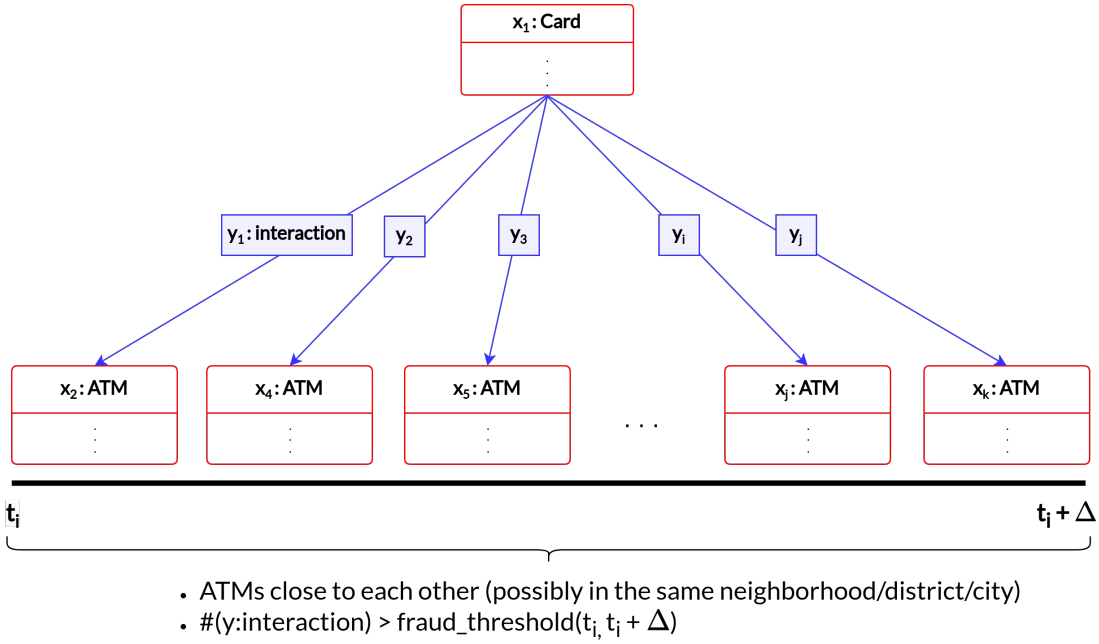


Figure 10: Property graph pattern - Lost-and-stolen card characterization. The interactions $y_1, ..., y_j$ depicted are of the *type* withdrawal

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 $t_i + \Delta = y_j.start$, 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 (in our case represented as the *withdrawal_day* `Card` entity property of our defined data model). Another indicator of this scenario to be considered could also be the *amount* value of the withdrawal operations performed, which, in these scenarios, is normally a low value to prevent that the card owner realises.

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.

4.3 Data Model

In the context of our work, the data we are considering can be seen as both *static* and *volatile* data, as we are considering a bank system application that, on the one hand, it contains all the *static* information related to it about the cards, clients, accounts, among others, that a bank typically gathers. And, on the other hand, it is a system where transactions made by clients with their cards at ATMs, POS terminals, etc., are continuously occurring and received in a streaming manner: the *volatile* data.

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, we decided to use the property graph data model (see 3.1). Our 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.

The property graph data model provides us many benefits: (i) it is a simple and easy method to represent entities and their relationships in the form of graphs; (ii) it is a convenient model to represent dynamic data sources, in our case the continuously occurring relations between cards and ATMs; (iii) and it provides us a direct way to model queries related with fraud pattern matching. Finally, mention that we used `Neo4j` (see 3.1) as the graph database management system to implement our property graph data model.

Design of the Property Graph Data Model

In what follows we describe the design of our property graph taken 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*, which is a fictional banking dataset publicly available in the Kaggle platform[20].

The proposed property graph data model is represented in Figure 11, consisting on both the stable and volatile property subgraphs merged. The main difference and the primary 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, with no incidence in any other element of the architecture. 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 and transaction links) and another, the stable, acting as a traditional bank database schema, from which to obtain the information details of the entities.

Stable Property Graph

Taking into account the reference dataset model, we designed a simplified version, as

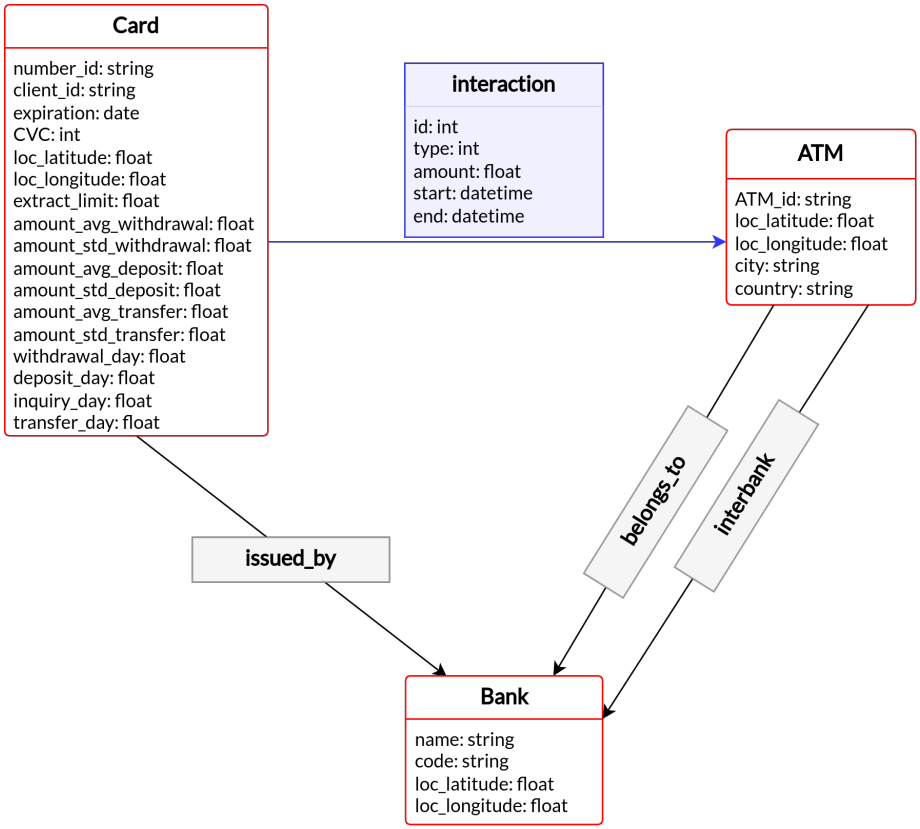


Figure 11: Complete property graph bank data model representation

shown in Figure 12, 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 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 13, 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.

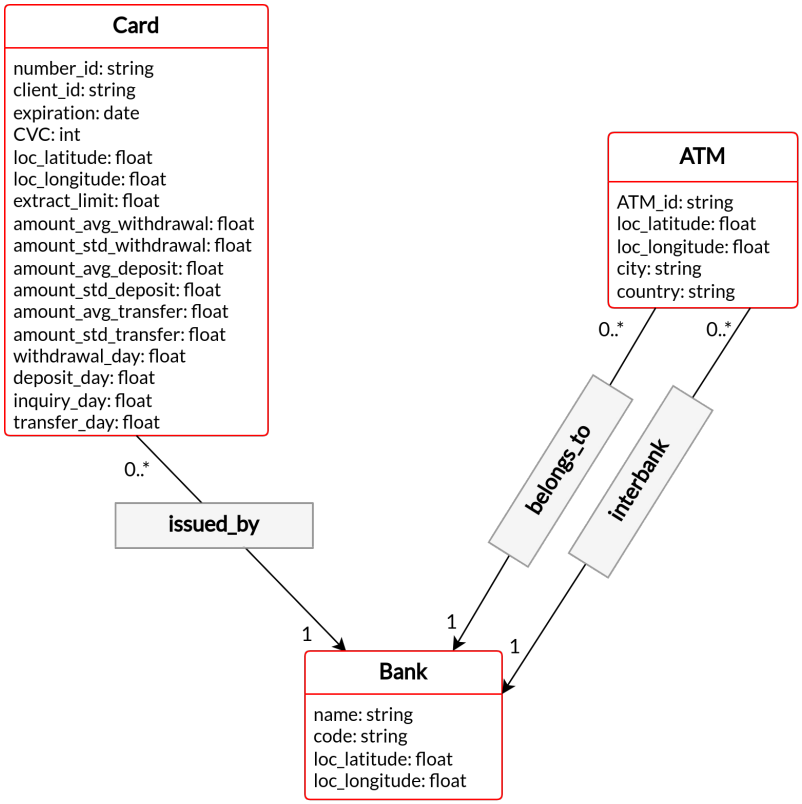


Figure 12: *Stable* property graph bank data model

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.

`Bank`

The `Bank` entity represents the bank we are considering in our system. Its properties consist on the bank *name*, its identifier *code* and the location of the bank headquarters, expressed in terms of *latitude* and *longitude* coordinates, as seen in Table 1.

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

Table 1: Bank node properties

`ATM`

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 POS terminal transactions could be included apart from the ATM ones. We distinguish two different kinds of ATMs, depending on their relation with the bank:

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

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

Property	**Description**
`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 properties consist on the ATM unique identifier *ATM_id*, its location, expressed in terms of *latitude* and *longitude* coordinates, and the *city* and *country* in which it is located, as seen in Table 2. Note that the last two properties are somehow redundant, considering that location coordinates are already included. In any case both properties are maintained since their inclusion provides a more explicit 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.

`Card`

The `Card` node represents the cards of the clients in the bank system. The `Card` node type properties, as depicted in Table 3, consist on the card unique identifier *number_id*, the associated client unique identifier *client_id*, the 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.

Property	**Description**
`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

In the proposed property graph bank data model 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 a card and a 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.

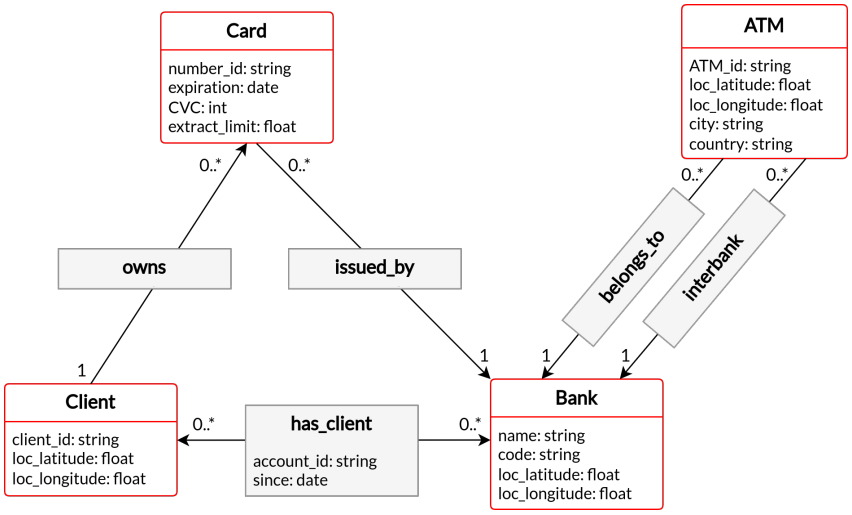


Figure 13: Alternative - A more complex stable property graph bank data model. It consists of four node entities: `Bank`, `ATM`, `Client` and `Card` with their respective properties, and the corresponding relationships between them. The relations are: a directed relationship from `Client` to `Card` `owns` 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.

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. These interactions are going to be continuously occurring and arriving to our system as data stream.

The proposed property graph, represented on Figure 14, is a subgraph of the original bank property graph model (Figure 11). 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. In addition, it contains the `interaction` relationship between the `Card` and the `ATM` nodes. The `interaction` relation contains as properties (see table 4): *id* as the interaction unique identifier, *type* which describes the type of the interaction (withdrawal, deposit, balance inquiry or transfer), *amount* describing the amount of money involved in the interaction in the local currency considered, and finally, *start* and *end* which define the interaction

datetime start and end moments, respectively.

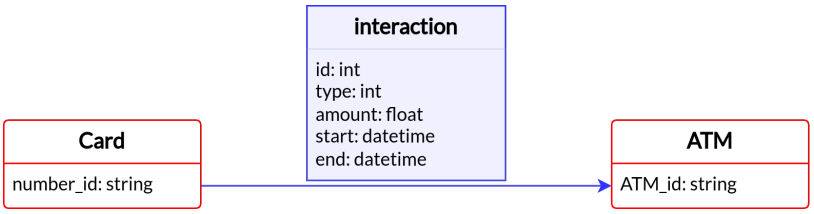


Figure 14: Volatile Property Graph Data Model

Property	Description
`id`	Interaction/Transaction unique identifier
`type`	Transaction type: withdrawal, deposit, balance inquiry or transfer
`amount`	Money amount involved in the transaction
`start`	Transaction start time moment
`end`	Transaction end time moment

Table 4: Interaction relation properties

A key aspect that we consider in our data model is the division of the `interaction` relation in two edges – the *opening* and the *closing* edges – both forming a single `interaction` relation. The *opening* edge (Figure 15) will be the indicator of the beginning of a new interaction between a Card and a 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 16) will indicate the end of the interaction, completing the values of the rest of the properties of the interaction: *end* and *amount*.

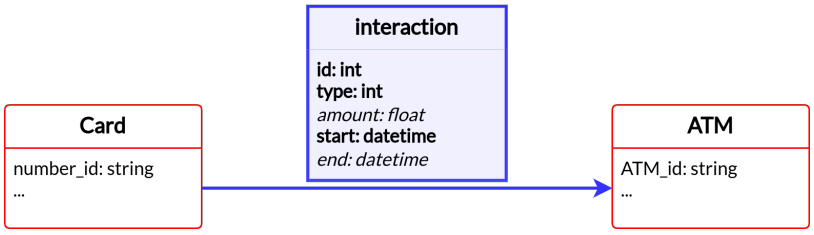


Figure 15: *Opening* interaction edge

With this division of the `interaction` relation in two edges we are simulating that for each transaction, our system receives an initial message when the transaction starts and a final message once the transaction is finished on the ATM. This can be a key aspect in our system, since it can allow 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 transaction detected has actually occurred.

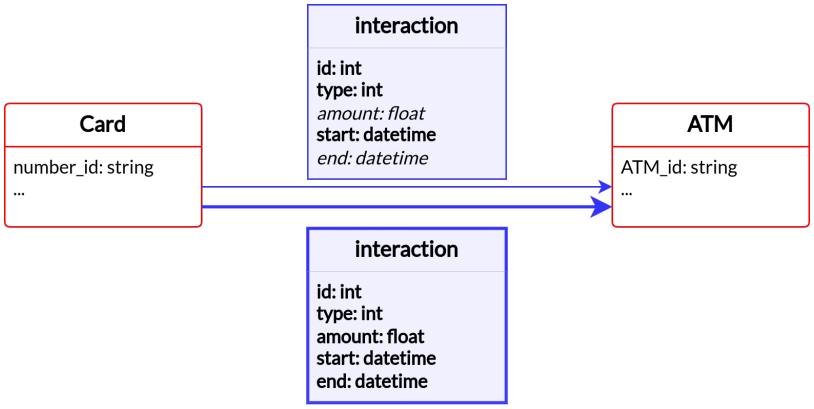


Figure 16: *Closing* interaction edge

4.4 The Query Model

Continuous Query Model

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 [11, 55]. 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 few 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 property graph, PG, representing a bank database. The bank database is continuously evolving due to the input ATM transactions data stream that it receives continuously. 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.

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 fraud graph pattern, the fraud graph pattern I (defined in 4.2), related with the card cloning characterization.

A first attempt to detect fraud pattern I in our setting is to compare, for each new transaction of a given card, its initial time against the final time of all the previous transactions of the same card. This procedure is given in Algorithm 1. Note that $\mathsf{S_c}$ refers to an individual node `Card c` subgraph of our volatile property graph. It stores all the card-ATM interactions $\mathsf{e_c}$ that are made with `Card c` for a considered time interval. $\mathsf{e_{new}}$ is the new incoming edge belonging to `Card c`, such that it is an opening interaction edge.

Note that this is this way since for the incoming closing interaction edges, we will not perform any fraud checking operation CheckFraud().

Algorithm 1 CheckFraud($\mathsf{S_c}, \mathsf{e_{new}}$) – **initial version**

Require: $\mathsf{S_c}$ refers to an individual node `Card c` subgraph of our volatile property graph. It stores all the card-ATM interactions $\mathsf{e_c}$ that are made with `Card c` for a considered time interval. The interactions are stored sorted by time.

Require: $\mathsf{e_{new}}$ is the new incoming edge belonging to `Card c`

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

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

- **Pairwise detection**. The checking of the anomalous fraud scenario is executed by doing the check between the new incoming edge $\mathsf{e_{new}}$ and each of the edges $\mathsf{e_i}$ of the `Card c` subgraph $\mathsf{S_c}$.
- **Backwards order checking**. The pairs ($\mathsf{e_{new}}, \mathsf{e_i}$) are checked in a backwards time traversal order of the edges of the $\mathsf{S_c}$ subgraph, 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 ($\mathsf{e_{new}}, \mathsf{e_i}$) is detected we stop the checking at this point and emit the corresponding alert. There is no need to continue the checking with previous edges of $\mathsf{S_c}$.
- **Emission of the pair** ($\mathsf{e_{new}}, \mathsf{e_i}$) **as the alert**. The alert is composed by the pair ($\mathsf{e_{new}}, \mathsf{e_i}$) 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 $\mathsf{e_i}$, which is previous to $\mathsf{e_{new}}$, in the case that $\mathsf{e_i}$ 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 $\mathsf{e_{new}}$ which is a regular transaction of the client. On the other hand, it can be $\mathsf{e_{new}}$, which is the last edge that arrived to the system, that directly causes the alert with the last (ordinary) transaction of the card.

However, a more detailed study, lead us to observe that Algorithm 1 has two important drawbacks. The first one is related to the amount of memory it requires since it has to keep stored all the transactions made with every card present in the system. The second one is related to the execution time since every new transaction of each card has to be compared

with all the previous transactions of the same card. To overcome the above mentioned disadvantages we propose Algorithm 2, a simplification of the initially proposed algorithm. There we just perform the checking between the new incoming edge e_{new} and the most recent edge of the subgraph S_c, e_{last}.

Algorithm 2 CheckFraud(S_c, e_{new}) – **definitive version**

Require: S_c refers to an individual node `Card c` subgraph of our volatile property graph. It stores all the card-ATM interactions e_c that are made with `Card c` for a considered time interval. The interactions are stored sorted by time.

Require: e_{new} is the new incoming edge belonging to `Card c`

1: $last \leftarrow |S|$
2: $e_{last} \leftarrow S[last]$
3: `t_min` $\leftarrow$ obtain_t_min(e_{last}, e_{new})
4: `t_diff` $\leftarrow e_{new}.start - e_{last}.end$
5: **if** `t_diff` $<$ `t_min` **then**
6: createAlert(e_{last}, e_{new})
7: **end if**

In what follows we argument the reason why it is sufficient to just check the fraud scenario among e_{new} and the last/most recent edge of the subgraph and not have to continue having to check with the rest of the edges.
Assume that we have a `Card c` subgraph S_c as the one depicted in Figure 17, and that we do not know if there have been anomalous scenarios produced between previous pairs of edges of the S_c and,

- Name $F_l(y_i, y_j)$ 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_c 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_c.

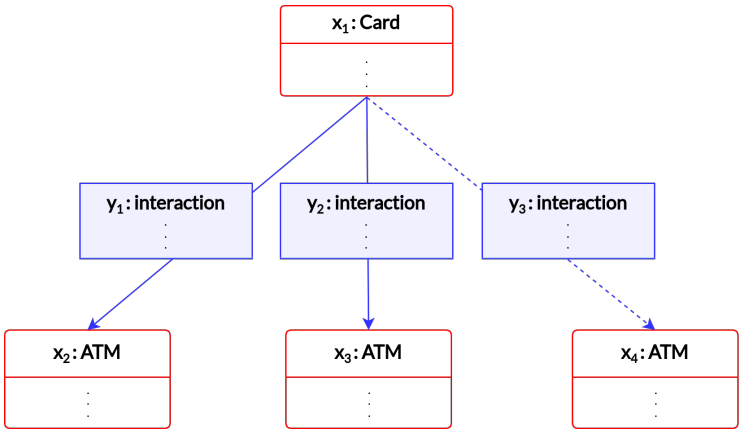


Figure 17: Example of a `Card c` subgraph S_c of the volatile property graph. S_c is an individual node `Card c` subgraph of our volatile property graph. It stores all the card-ATM interactions e_c that are made with `Card c` for a considered time interval. The interactions are stored sorted by time.

Note that, given the above description, we can either have that:

- $\mathsf{F_I}(\mathsf{y_2}, \mathsf{y_3})$. Then, we emit an alert of this anomalous scenario produced between the pair $(\mathsf{y_2}, \mathsf{y_3})$. We could continue checking for anomalous scenarios between $\mathsf{y_3}$ 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.
- $\neg\mathsf{F_I}(\mathsf{y_2}, \mathsf{y_3})$. In this case 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:
  - $\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_2})$. Having this it can happen that either $\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_3})$ or $\neg\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_3})$. In the case of $\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_3})$, since $\neg\mathsf{F_I}(\mathsf{y_2}, \mathsf{y_3})$, we can infer that the anomalous scenario detected between $\mathsf{y_1}$ and $\mathsf{y_3}$ is a continuation of the same previous anomalous scenario detected between $\mathsf{y_1}$ and $\mathsf{y_2}$. Therefore, we can conclude that this does not constitute a new anomalous scenario that would require an alert.
  - $\neg\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_2})$. It can be shown that *by transitivity*, having $\neg\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_2}) \wedge \neg\mathsf{F_I}(\mathsf{y_2}, \mathsf{y_3}) \implies \neg\mathsf{F_I}(\mathsf{y_1}, \mathsf{y_3})$.

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

So, we can state that, in an implementation of the detection of the fraud pattern I, for a subgraph $\mathsf{S_c}$, it would be sufficient to only store the last incoming transaction edge $\mathsf{e_{last}}$.

Finally, note that due to this algorithmic method, a fraud pattern I alert could be triggered both when an anomalous interaction arrives, $\mathsf{e_{anom}}$, with respect to its previous transaction $\mathsf{e_{prev}}$, emitting the alert pair $(\mathsf{e_{prev}}, \mathsf{e_{anom}})$; and also when the subsequent transaction $\mathsf{e_{next}}$ arrives, with respect to the anomalous, emitting the alert pair $(\mathsf{e_{anom}}, \mathsf{e_{next}})$.

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

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 Approach DPA, the $\mathsf{DP}_{\mathsf{ATM}}$, where, by definition, its architectural framework gets defined as a DP.

The primary 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.

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. So far, for the continuous query considered, we have only implemented a one-time-based infinite window. This means that once a card starts to be tracked in our system it will remain on it infinitely, even if no transactions are produced for a long time. However, exploring more advanced time window policies could be beneficial, particularly for other types of continuous queries. We leave this as future work.

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*. Regarding the behavior of 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.

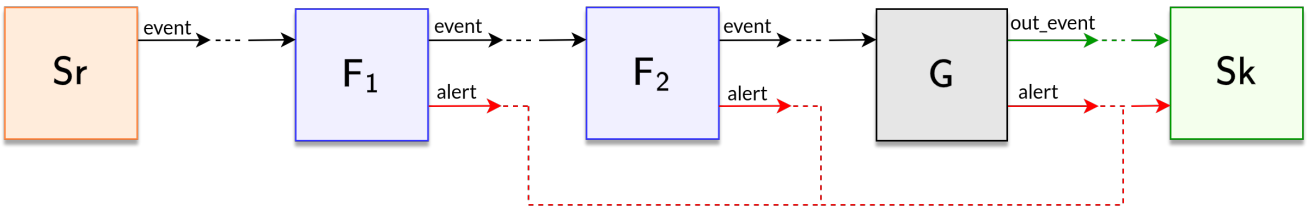


Figure 18: Example of the pipeline schema of a DP_{ATM} instance, with the different stages S_r, F_1, F_2, G and S_k and the communication channels: event (represented as the black arrows) and alert (represented in red color arrows). out_event (in green) represents a direct communication event channel between G and Sk.

More detail regarding each of the stages behavior is provided next. An example of the pipeline schema of a DP_{ATM} instance is shown in Figure 18.

- *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 4.3. 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 $V_F = \{v_1, \ldots, v_k\}$, representing the Cards being tracked by F. Therefore, each root represents a Card c where v_i is the Card property *number_id* value: $v_i = c.number\_id$. 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 $|V_F| \leq maxFilterSize$.

  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 ($e.number\_id$), that is when $v_i = e.number\_id$. This means that F is currently tracking the activity of the Card c to which the interaction e belongs.

  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: $|V_F| < maxFilterSize$. Therefore, card c is introduced as a new root parameter $v_{k+1} = e.number\_id$ to V_F: $V_F = V_F \cup v_{k+1}$. These two belonging conditions are summarized as:

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

  Otherwise ($\nexists v_i \in V_F$ such that $v_i = e.number\_id \wedge |V_F| = maxFilterSize$), 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 = e.*number_id*, and then perform the algorithm to test (each of) the continuous query pattern(s) with their associated card c subgraph(s).

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 differ. 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.

- *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.

DP$_{ATM}$ - Implementation

The implementation of the proof of concept can be found in Github [14]. 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 [34, 53, 43], which is the computing technique that we need to implement the DP schema for our DP$_{ATM}$ 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 [17] 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 [32] to easily interact with a Neo4j graph database instance through a Go application. More details regarding the connection are later explained.

Neo4j Connection

Our DP$_{ATM}$ 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 [33]. 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 [31, 32].

In our implementation of the DP$_{ATM}$ 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 query it.

- **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 [21] 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 application protocol [28], 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 1: 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.

As a remark, note that 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. In any case, it is important to remark that many multiple parallel sessions may cause overhead to the database. This could be the case for the architectural design we are proposing, where we have a session per each *Filter* stage. Consequences of this are expected to be reflected in the experimental analysis.

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 2: `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 4.3).

```
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 3: Data type for the interaction edges in Go

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

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

Listing 4: `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 part of 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.

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

Listing 5: `Alert` data type

- out_event: direct dedicated `event` channel between the G and Sk.
- internal_edge: Internal `event` channel between a Fstage 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 DP$_{ATM}$. Each stage takes the form of a `goroutine` of the `Go` language. A final picture of an example of a pipeline schema of a DP$_{ATM}$ instance is shown in Figure 19.

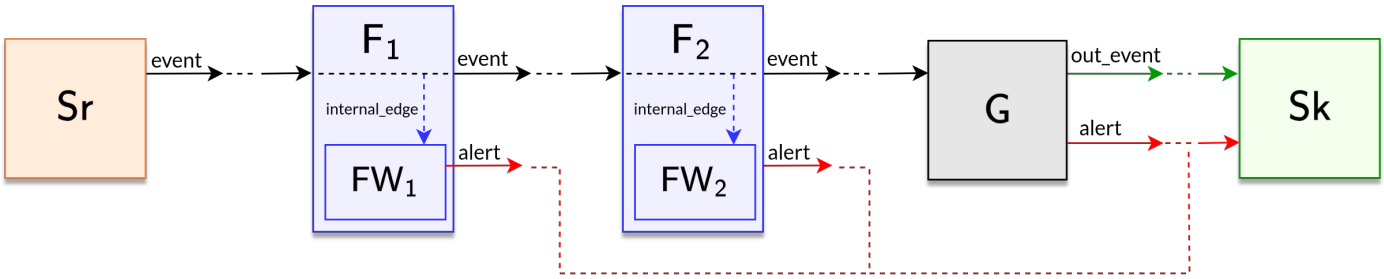


Figure 19: Example of the pipeline schema of a DP$_{ATM}$ instance, with the different stages S_r, F_1, F_2 and its respective substages FW_1, FW_2; G and S_k and the communication channels: event (represented as the black arrows) and alert (represented in red color arrows). out_event (in green) represents a direct communication event channel between G and Sk.

Source

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

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 DP$_{ATM}$ 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 6, we implemented two different cases of simulations. The real-case and the high load stress 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. For this, 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 6: `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 chunk size (*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 6.5 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.

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 4.4) 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 7: 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.

- **Decoupled event handling**. We implement a *Decoupled Event Handling*, as in the DP implemented on [13]. 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 *mail-box*, 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 fully dedicated to do the corresponding processing, the continuous query evaluation, 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. Figure 20 shows a detailed representation of this F and FW stages.

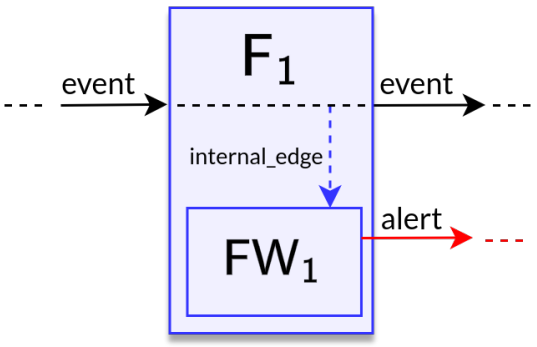


Figure 20: Representation of a *Filter* F_1 and its corresponding *Filter Worker* FW_1 pipeline stage

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 [52, 35].

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.

- **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 soon became obvious that this 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 6.4.1, it became clear the need of allowing to share each *Filter* stage for multiple bank cards. Formally, each *Filter* F was implemented to be able to hold a certain subset of root parameters $V_F = \{v_1, \ldots, v_k\}$, 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. 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, `e` $\in$ 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 8: Hash tables `map` data structures on a *Filter* F stage in `Go`

One could think of using one single `map` data structure to do the check `e` $\in$ 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 `e` $\in$ 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 not defined what happens (possible race conditions) in `Go` in this situation. 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 2. In the algorithm 3 we provide a more detailed description of its implementation. 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 $\mathsf{S_c}$ and the interaction `Edge` $\mathsf{e_{new}}$ of the `EdgeStart` event. It is checked that there exists a sufficient time distance between $\mathsf{e_{new}}$ and $\mathsf{e_{last}}$; the previous added edge to Sc before $\mathsf{e_{new}}$. To do it, we need to obtain the minimum needed time `t_min` to traverse the distance between the ATMs of the $\mathsf{e_{new}}$ and $\mathsf{e_{last}}$ interactions: $\mathsf{ATM_{last}}$ and $\mathsf{ATM_{new}}$, corresponding to the ATMs with identifiers $\mathsf{e_{last}}$.`ATM_id` and $\mathsf{e_{new}}$.`ATM_id`, respectively. `t_min` is obtained through the function call obtainTmin($\mathsf{e_{last}}, \mathsf{e_{new}}$) on line 7. 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 9). 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 $\mathsf{DP}_{\mathsf{ATM}}$ implementation.

Algorithm 3 `checkFraud`($\mathsf{S_c}, \mathsf{e_{new}}$)

Require: $\mathsf{S_c}$ is a non-empty subgraph of interaction edges of card `c`, $\mathsf{e_{new}}$ is the `Edge` related with the new incoming opening interaction `EdgeStart` of card `c`

1: $\mathsf{e_{last}} \leftarrow S_c[|S_c| - 1]$ {Retrieve the last edge from the subgraph $\mathsf{S_c}$}
2: **if** $\mathsf{e_{last}}$.`Tx_end` is empty **then**
3: `LOG: Warning: A tx (`$\mathsf{e_{new}}$`) starts before the previous (`$\mathsf{e_{last}}$`) ends!`
4: **return**
5: **end if**
6: **if** $\mathsf{e_{last}}$.`ATM_id` $\neq$ $\mathsf{e_{new}}$.`ATM_id` **then**
7: `t_min` $\leftarrow$ obtainTmin($\mathsf{e_{last}}, \mathsf{e_{new}}$)
8: `t_diff` $\leftarrow$ $\mathsf{e_{new}}$.`Tx_start` $-$ $\mathsf{e_{last}}$.`Tx_end`
9: **if** `t_diff` $<$ `t_min` **then**
10: emitAlert($\mathsf{e_{last}}, \mathsf{e_{new}}$)
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 9: 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:

- $\mathsf{coords}_{\mathsf{last}} = (\mathsf{ATM}_{\mathsf{last}}.\texttt{loc\_latitude}, \mathsf{ATM}_{\mathsf{last}}.\texttt{loc\_logitude})$
- $\mathsf{coords}_{\mathsf{new}} = (\mathsf{ATM}_{\mathsf{new}}.\texttt{loc\_latitude}, \mathsf{ATM}_{\mathsf{new}}.\texttt{loc\_logitude})$

then `t_min` is calculated as: $\texttt{t\_min} = \texttt{distance}(\mathsf{coords}_{\mathsf{last}}, \mathsf{coords}_{\mathsf{new}})/\ \texttt{maxSpeed}$. $\texttt{distance}(\mathsf{coords}_{\mathsf{last}}, \mathsf{coords}_{\mathsf{new}})$ is the *great-circle* distance or *orthodromic* distance between the two coordinate points. It is calculated using the `Go haversine package` [46]. `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 $\texttt{maxSpeed} = 500$ 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($\mathsf{e}_{\mathsf{last}}, \mathsf{e}_{\mathsf{new}}$) on line 10. The `Alert` will contain the subgraph built with the two interaction edges causing the alert: $\mathsf{e}_{\mathsf{new}}$ and $\mathsf{e}_{\mathsf{last}}$ 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 implementation of this function is shown in `Go` in the 10 code listing. Note that out_alert refers to the output alert channel that connects the *Filter* stage with the *Sink* stage (`chan<- cmn.Alert`).

```
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 10: 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 11. In addition we show on Figure 21 a representation of a *Filter* and its corresponding *Filter Worker*. On it we can see the different functions assigned to each; *Filter* acts as a *mailbox* deciding whether an interaction edge belongs to it or not; whereas *Filter Worker* is the responsible of managing the multiple card subgraphs and perform the continuous query evaluations.

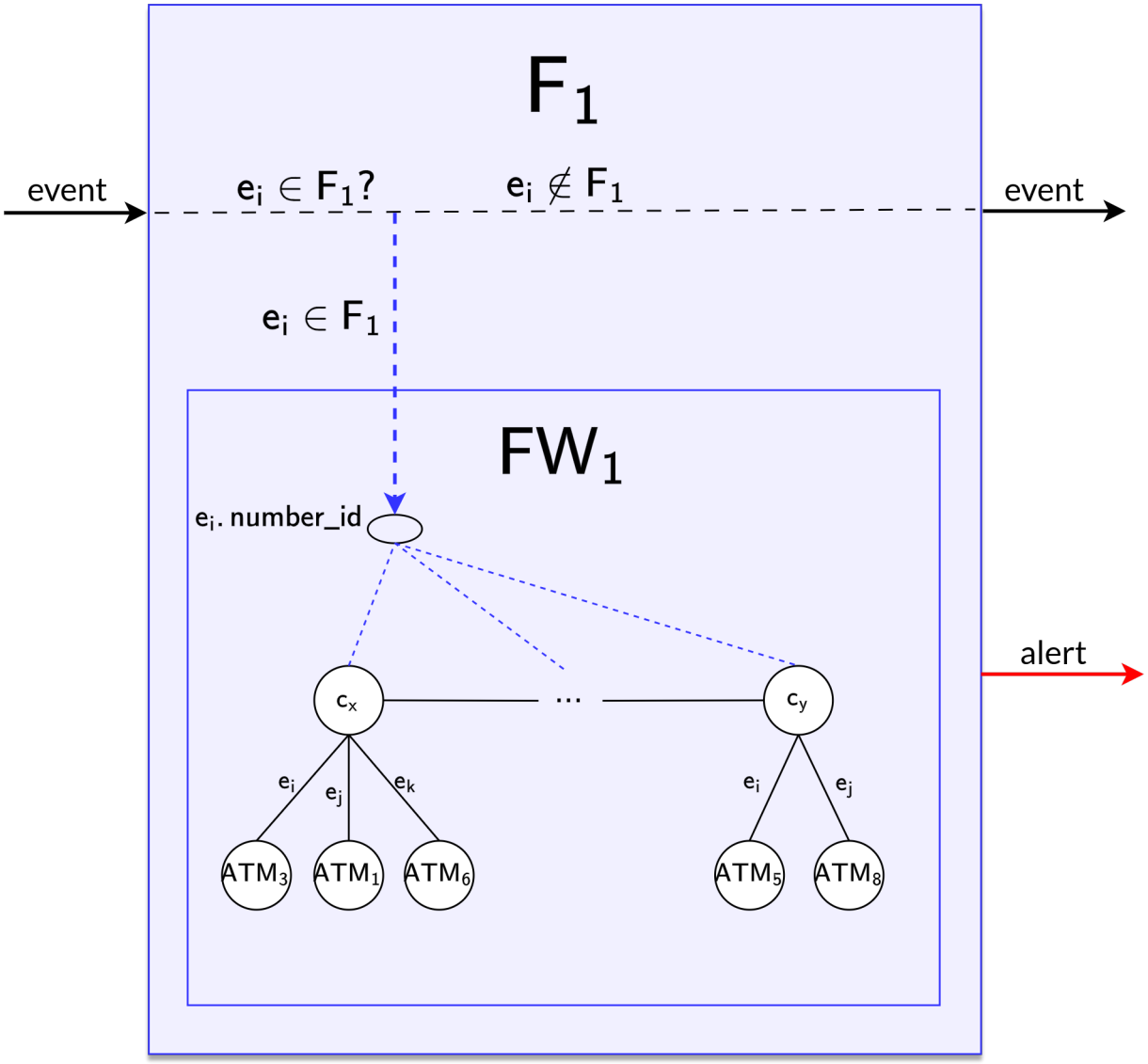


Figure 21: Representation of a *Filter* and its corresponding *Filter Worker*. On it we can see the different functions assigned to each; *Filter* acts as a *mailbox* deciding whether an interaction edge belongs to it or not; whereas *Filter Worker* is the responsible of managing the multiple card subgraphs and perform the continuous query evaluations. event is represented with black arrows, alert with red color arrows and internal_edge is represented with a blue dotted arrow.

```
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 11: 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.

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 from the event channel.

6 Experiments

In this section we provide all the details regarding the experimental evaluation done in order to show the performance of the $\mathsf{DP}_{\mathsf{ATM}}$ system. 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.

First, in 6.1 we describe the design of the experiments that we proposed to study the performance of the $\mathsf{DP}_{\mathsf{ATM}}$ system in different scenarios. Then, in Subsection 6.2 we describe the running environment.

As we already mentioned 3, the system performance is evaluated using several standard metrics namely: (i) the response time (`RT`) and the average time (`MRT`) that measure the exact and average time that the system takes to output a check result, from the moment in which the transaction enters the system until the check result is given; (ii) the execution time (`ET`) that states how fast is the system to process the whole input data stream; (iii) the `dief@t`, that indicates the continuous behavior of the system throughout the execution time and (iv) the throughput (`T`) (number of results and of interactions emitted by second). A detail description of the evaluation metrics is given in 3.3.

As mentioned before, due to the sensitive nature of the bank data, to carry our experiments we needed to generate our own synthetic datasets, for both the bank database and the stream of transactions. In Subsection 6.3 we provide full detail on the generation of our own synthetic bank databases and, regarding the transaction stream, we devote the Subsection 6.4.1, to detail the considered transaction streams for our experiments and explain how we performed their generation. Finally, in Subsection 6.5 we discuss different considered methods to consume the stream of transactions, and the empirical comparisons from which we decided the method of stream consumption by the $\mathsf{DP}_{\mathsf{ATM}}$ in our experiments. Part of this discussion was already advanced in the description of the implementation of the *Source* Sr stage in Section 5.

In the empirical study we conducted we considered an alternative *result* definition. In effect, for our experimental purposes we considered that *results* emitted by the $\mathsf{DP}_{\mathsf{ATM}}$ correspond to *fraud pattern checking*. That is, all the fraud pattern checking are considered as results on this definition, even if they are negative, i.e. when they do not derive in the creation of an alert. Considering all the fraud pattern checking as results is done in order to better analyze the continuous production of results by the $\mathsf{DP}_{\mathsf{ATM}}$, reflecting all the fraud pattern checking that the system is undergoing. Note that, considering all the checking as results induces a communication overhead among *Filter* F stages and the *Sink* Sk stage, where the results are gathered. In a production version of $\mathsf{DP}_{\mathsf{ATM}}$ we would use the original definition of a $\mathsf{DP}_{\mathsf{ATM}}$ system result, i.e. only alerts would be considered results, since sending negative fraud pattern checks to Sk would be of no interest. This means that in a production setting, the overhead in the communication channels among the F's and Sk and the corresponding processing of results in Sk would be reduced at maximum.

6.1 Design of Experiments

To evaluate the performance of the $\mathsf{DP}_{\mathsf{ATM}}$ we initially proposed two kinds of experiments. On the one hand, the evaluation of the $\mathsf{DP}_{\mathsf{ATM}}$ in scenarios simulating close-to-reality transaction frequencies (E0). On the other 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.

6.1.1 E0: Evaluation in a Real-World Stress Scenario

This experiment was proposed to evaluate the behavior of the $\mathsf{DP}_{\mathsf{ATM}}$ in a close-to-reality scenario in terms of the frequency of transactions of the input stream that arrive to the system.

In a real case scenario, the frequency of the transactions on the transaction stream reaching our system will depend mainly on the size of the bank considered. For small-sized banks we do not expect this transaction frequency to be quite high due to its reduced number of clients - and therefore cards - that can perform operations on a certain period of time. On the contrary, for big sized banks we do expect the frequency of transactions to be higher. In general, we assume that, on average, a cardholder does not perform more than one transaction per day at an ATM. This is indeed the case for the transaction streams that we are going to employ in our experiments, on them the average number of ATM operations per client per day is 0.666 (see Table 5 on 6.3).

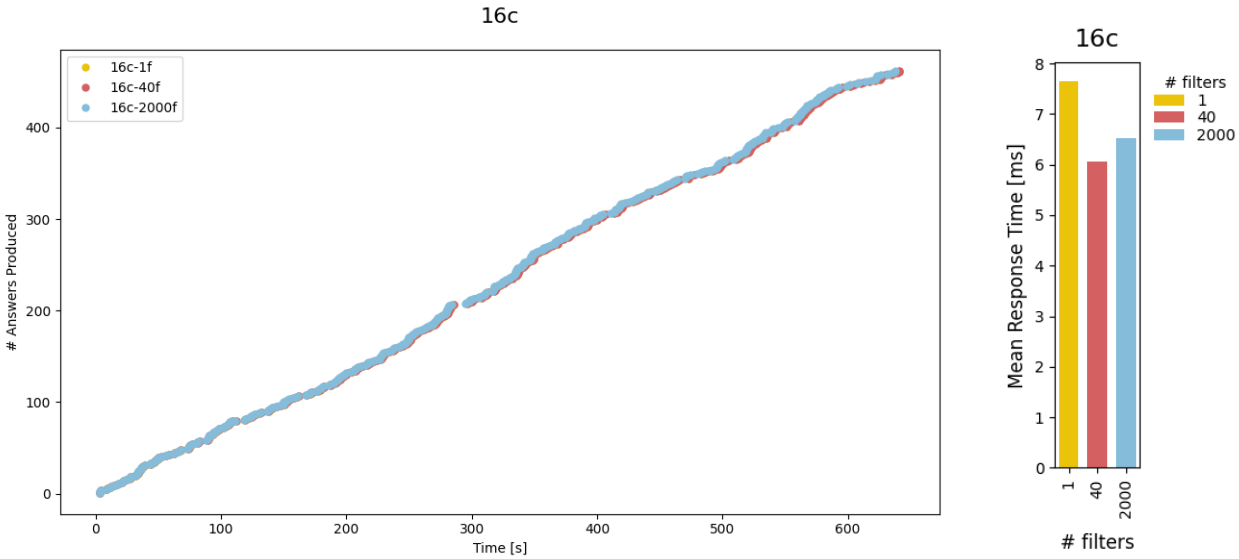
To simulate the arrival of transactions with their real frequency, we will feed the system with the transaction stream replicating their actual arrival times. This means maintaining the appropriate time gaps between consecutive transactions. Specifically, for each consecutive pair of interactions $(\mathsf{e}_\mathsf{i}, \mathsf{e}_{\mathsf{i}+1})$, the serving of the second interaction $\mathsf{e}_{\mathsf{i}+1}$ to the pipeline will be delayed by the exact time difference from the previous interaction e_i.

For this experiment, we initially considered a small bank database system, GDB_A, consisting of 2000 cardholders and 50 ATMs. Considering that each cardholder, on average, does not perform more than one ATM operation per day, we decided to simulate a transaction stream spanning several weeks or months to achieve a reasonable stream size. However, two issues arose: first, performing a single simulation spanning multiple days or months proved impractical due to time constraints; second, we realized that such a low transaction frequency would not yield meaningful insights into the system's performance due to insufficient stress. To address these challenges, we considered two possible solutions:

- (i) scaling the transaction stream by mapping it into an equivalent but smaller-sized stream
- (ii) considering a larger bank database system

We initially applied the *scaling* workaround through some experiments on the GDB_A database system, with the GDB_A-30 stream, a transaction stream representing 30 days of simulated activity for the GDB_A bank. We tested different time scalings, reducing the stream to 12, 6 and 1 hour length. However, we needed to further adapt the scaling by adjusting transaction timestamps to the order of microseconds μs, as the system was failing to report some expected alerts under the original to the second s scaling. Finally, with this scaling some experiments with the same GDB_A-30 stream were carried; in particular with a 10 minute length scaling. On Figure 22 we can see some results of this test. We can observe that the differences between the different $\mathsf{DP}_{\mathsf{ATM}}$ configurations tested with 1, 40 and 2000 filters are almost negligible. On 22a the result traces of all the variants are almost identical. And in 22b the differences in the `MRT` are also insignificant, in the order

of less than 1 or 2 milliseconds. This was due to the still low level of stress to which we were subjecting the $\mathsf{DP}_{\mathsf{ATM}}$.



(a) (b)

Figure 22: For 1, 40 and 2000 number of filters configurations of the $\mathsf{DP}_{\mathsf{ATM}}$ for the $\mathsf{GDB}_{\mathsf{A}}$-30 input stream test of the $\mathsf{GDB}_{\mathsf{A}}$. Run with 16 cores. (a) Shows the results trace of pairs (result, time(s)). Vertical axis shows the *result* number and horizontal axis shows the time in seconds at which that result is produced. (b) Shows the MRT achieved for the different configurations. On this test, we considered only the alerts as system *results*.

In summary, we realized that we needed to test the system under a high-load stress scenario (E1), forcing the transaction stream frequency to be as high as possible in order to identify the system's limits.

6.1.2 E1: Evaluation in a High-Load Stress Scenario

This experiment aims to serve as a means to evaluate the behavior of the $\mathsf{DP}_{\mathsf{ATM}}$ in a high load stress scenario. This scenario is a worst case scenario in terms of the frequency of transactions on the input stream arriving to the system, which will be simulated to be at its maximum peak.

In a real case scenario, the frequency of the transactions on the transaction stream reaching our system will depend mainly on the size of the bank considered. For small-sized banks we do not expect this transaction frequency to be quite high due to its reduced number of clients - and therefore cards - that can perform operations on a certain period of time. On the contrary, for big sized banks we do expect the frequency of transactions to be higher. With this, we can claim that this experiment is able to show what would be a real case scenario for a large bank system, where the frequency of transactions occurring in a certain time period is quite high. However, considering that, we can assume that, on average, a cardholder does not perform more than one transaction per day at an ATM - in our considered transactions streams the average number of ATM operations per client per day is 0.666 (see Table 5) - it is important to remark that this scenario would be an

unrealistic scenario for most bank systems and that in reality we would expect a lower frequency of transaction input, allowing the $\mathsf{DP}_{\mathsf{ATM}}$ to exhibit a better performance.

We will simulate it by providing the $\mathsf{DP}_{\mathsf{ATM}}$ system the transactions of the stream immediately one after the other, as fast as possible. This experiment will be carried considering two bank databases: $\mathsf{GDB}_{\mathsf{A}}$, simulating a small bank database system; and $\mathsf{GDB}_{\mathsf{B}}$, simulating a large bank database system. In both cases the stream of transactions is provided at the same speed, the difference lies in the number of cards that each system contains. This will make the $\mathsf{DP}_{\mathsf{ATM}}$ for the large bank system be a *larger* $\mathsf{DP}_{\mathsf{ATM}}$, by having to track the activity of more cards. We also expect the time needed to query the stable bank database to be higher in the large bank system.

To evaluate the behavior of the system in this scenario, it is essential to identify the metrics that can tell us how good the system behaves on it. As a real-time system, a key functionality is to be able to emit a response as fast as possible. In our case, to minimize the time to alert the bank system or a user of a potential fraud being produced, being able to minimize the derived damages. A really important metric to measure this is the Response Time, `RT`, and its average value, the Mean Response Time, `MRT`. Also relevant is the evolution of the `RT` through time, and the evaluation of the continuous behavior of the system through the tested time, measured with the `dief@t` and `dief@k` *diefficiency* metrics. Finally, other classical metrics are considered to evaluate the speed at which the system can process a full slice of a stream of transactions: the throughput, `T`, in terms of answers emitted per unit of time and the execution time, `ET`, to process the full stream.

We considered different $\mathsf{DP}_{\mathsf{ATM}}$ configurations regarding the number of filters with which we construct the DP pipeline, and we compare the different DP versions against a sequential baseline program. These systems were tested for different resources variations, in terms of the dedicated number of cores on which we run them. The description of the results of the experiments performed is detailed on 7.

6.2 Experimental Settings

6.2.1 Hardware

The experiments were run in the machine nodes of the RDLab-UPC cluster at `https://rdlab.cs.upc.edu/` [38]. These nodes have different processors: Intel(R) Xeon(R) CPU X5675 @ 3.07GHz and 12 cores, Intel(R) Xeon(R) CPU X5670 @ 2.93GHz and 12 cores, Intel(R) Xeon(R) CPU X5660 @ 2.80GHz and 12 cores, Intel(R) Xeon(R) CPU X5550 @ 2.67GHz and 8 cores, and Intel(R) Xeon(R) CPU E5-2450 @ 2.50GHz and 16 cores. We restricted to 16GB of RAM the maximum memory requested to run each of the experiments and variate the number of requested cores depending on the experiment. The timeout of each experiment was 24h.

Regarding the graph database we used a Neo4j `5.22.0` (community edition) installed on a virtual machine with 4 cores and 20GB of RAM, accessible for the cluster nodes through a *Bolt* protocol on `neo4j://10.4.30.227:7687`.

6.2.2 Software

We used the version `1.20` of the `Go` language to implement the $\mathsf{DP}_{\mathsf{ATM}}$ system. The connection to the Neo4j graph database instance was done using the version `v5.24.0` of the `Neo4j Go driver` [33].

Full detail regarding the software used for the implementation of the $\mathsf{DP}_{\mathsf{ATM}}$ system is given in the $\mathsf{DP}_{\mathsf{ATM}}$-Implementation subsection of Section 5.

6.3 Datasets

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 datasets, comprising the synthetic stable property graph bank database and the streams of synthetic transactions.

For this, we utilized the synthetic *Wisabi Bank Dataset* [20] as a base for the construction of our own datasets. This dataset could have been adapted to our data model definition and being used it for our experiments. However, in order to have full flexibility, and not being constrained to this unique dataset, we decided to develop a tool to be able to generate customizable datasets adapted to any experimental needs.

Wisabi Bank Dataset. The *Wisabi Bank Dataset* is a fictional banking dataset publicly available in the Kaggle platform. We used it as a first base to do general customizable 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 dataset and contains card-ATM transactions. Additionally, it provides good heterogeneity on the different kind of transactions: withdrawals, deposits, balance inquiries and transfers. Regarding this operations, we gathered the average number of the different kinds of operations performed per cardholder on a day in table 5. More 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).

Operation	Value
Total operations per day	0.6660
Withdrawals per day	0.3696
Deposits per day	0.0742
Inquiries per day	0.0743
Transfers per day	0.1478

Table 5: Wisabi bank dataset average number of the different kinds of operations performed per cardholder on a day

The dataset consists on ten *csv* tables each with different information which is summarized on Table 6.

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 6: 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.

6.3.1 Synthetic Bank Database Creation

To do the creation of our synthetic bank database, first we describe the creation of the *csv* files that conform a bank dataset and then we describe how the creation and population of the graph database in Neo4j is done.

Bank Dataset Creation: `bankDataGenerator.py`

To generate a 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 needed to construct a bank graph database. 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 of the *Wisabi Bank Dataset* (publicly available on Kaggle [20]).

2. Ensure to have the `behavior.csv` file or run `$> python3 behavior.py` to create it. This creates a *csv* file with the gathered customer behavior properties from this dataset.

3. Run `$> python3 bankDataGenerator.py` 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 the bank dataset entities, as defined in 4.3.

Bank: `bank.csv`

Since a unique bank instance is considered, the values of the properties of the bank node are manually assigned, leaving them completely customizable (see an example on 12).

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

Listing 12: Example of a bank.csv

ATM: `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 13.

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

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

- **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 ATMs. 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, and so on). 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 13: Example of atm.csv

Card: `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 table 8.

- **Card and client identifiers**:

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

- ***expiration* and *CVC* properties**: they are not relevant, could be empty value properties indeed or a same toy value for all the cards. The same values are given for all the cards: $expiration = 2050\text{-}01\text{-}17$, $CVC = 999$.

- **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 7. 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 - considered 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 7: *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 would 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.

$$extract\_limit : amount\_avg\_withdrawal * 5$$

```
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

amount_avg_withdrawal,amount_std_withdrawal,withdrawal_day
24318.18,28174.96,0.2411

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

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

Table 8: Example of a Card instance record on the `card.csv` file

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 14.

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

Listing 14: 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 15.

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

Listing 15: 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 16.

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

Listing 16: Example of a card-bank.csv

Bank Database Population

To populate the Neo4j graph database instance with the generated bank dataset, we developed the `populatemodule Go` module. This module takes the bank dataset in the form of the generated *csv* files and populates the indicated Neo4j graph database instance.

Two different population modules are proposed. The first, `csvimport`, does it by directly importing the *csv* files using the `Cypher`'s `LOAD CSV` command [29], while the second method, `cypherimport`, does it by parsing the *csv* data and running the creation of the nodes and relationships using `Cypher` directives. The first method is more convenient

whenever we are able to access the file system of the machine in which the Neo4j graph database instance is hosted. Otherwise, the second method provides us an alternative way not needing to access the file system of the machine hosting the Neo4j instance. The module tree structure is depicted in Figure 23. 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.

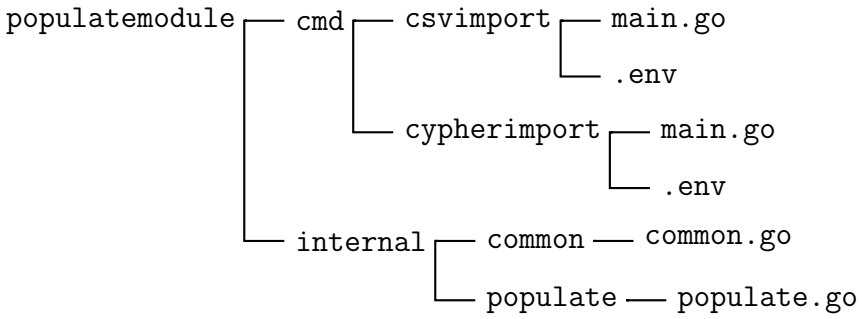


Figure 23: `populatemodule` file structure

To indicate the Neo4j graph database instance to be populated we need to first set up correctly the `.env` file (see the listing 17) located inside the desired method subdirectory, where we have to define the corresponding Neo4j environment variables *URI*, *username* and *password* to access the Neo4j graph database instance.

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

Listing 17: Example of a `.env` file, from which the connection credentials will be gathered by our `populatemodule`.

Both methods, before doing the population of the Neo4j instance with the *csv* files, perform the creation of 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 18: Uniqueness ID constraints

Finally, we include a brief description on how to run each of the described population methods of the module:

- `csvimport`: Cypher's `LOAD CSV`
  We use the `LOAD CSV` clause which directly allows to load *csv*'s into Neo4j, creating the nodes and relations expressed on the *csv* files of the dataset. To use this method 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`

- `cypherimport`: *Manual* `Go` parsing
  On this method we parse the data of the *csv* files in our `Go` program, and then, for each record we run a `Cypher` command to import it as a node instance or relationship in the Neo4j database. This method allow us to avoid locating the *csv* dataset files on the machine where the Neo4j graph database instance is running (in the case it runs in a different machine) To use this method do:

  – Run `$> go run populatemodule/cmd/cypherimport/main.go <csvPath>` providing as argument `<csvPath>` the path of the directory where the *csv* dataset files are located.

6.3.2 Synthetic Transaction Stream 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 4.3) 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 card-ATM bank transaction stream.

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 scenario, whereas the anomalous transaction set is composed of the *irregular/anomalous* transactions that are intentionally created to produce the 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 regular transaction stream 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. The union of both sets will form the final generated transaction stream.

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. So far, since we are only considering the fraud pattern I for our proof of concept system, the constraints that we need to impose on the generation of the regular transaction set are:

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

As a summary of the regular transaction set generation method that we developed, we provide a pseudocode on the Algorithm 4. Some of its parts are later explained in detail.

The generation of the transaction stream is performed for a customizable `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 gathered behavior (see table 7) to determine the number and type of interactions performed in the defined days time interval: [`START_DATE`, `START_DATE` + `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, `residence_loc`, of the client of the card, represented 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 [`START_DATE`, `START_DATE` + `NUM_DAYS`] respecting the constraint related with the fraud pattern I.

Algorithm 4 Regular Transactions Generation

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

In what follows we give full detail of each of the steps of the Algorithm 4, to complete the explanation of the regular transaction set generation method.

1. **Selection of ATMs**: `ATM_subset`, $\overline{\texttt{ATM\_subset}}$ ← createATMsubset(`residence_loc`)

   `ATM_subset` is the subset of ATMs of the stable bank dataset. It will consist of the ATMs in which we will allow the interactions of a card to occur. We set a limit on the size of this subset, considering only a maximum ratio of the total number of

ATMs of the bank network (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:

- ***Neighborhood* location selection**: Only the closest |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* / belonging to the neighborhood of 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 is imposed:

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

- **Random selection**: The ATM_subset is built by randomly selecting |ATM_subset| ATMs from the considered bank network.

2. **Calculate minimum time distance (t_min_subset)**:
t_min_subset ← calculate_t_min_subset(ATM_subset)

   t_min_subset is the minimum time difference 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 km at an assumed speed that any two locations can be traveled in the case of regular transaction scenarios: REGULAR_SPEED km/h.

3. **Select the number of transactions num_tx**: num_tx ← decide_num_tx()

   Based on the behavior of the card, we decide the number of transactions num_tx to generate for the card for the defined days time interval [START_DATE, START_DATE + NUM_DAYS] using a Poisson distribution as:

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

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

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

4. **Time distribution of the `num_tx` transactions**: $T \leftarrow \text{distribute}(\texttt{num\_tx}, \texttt{t\_min\_subset})$

   Along the selected time interval $[\texttt{START\_DATE}, \texttt{START\_DATE} + \texttt{NUM\_DAYS}]$ we do a random uniform distribution of the `num_tx` transactions. T contains the list of all the `start` and `end` times tuples for each of the `num_tx` transactions, respecting the constraint I: ensuring that, for the considered card, 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:

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

5. **Assignment of the remaining properties' values:**
   Once all the previous steps are done, the remaining values for the properties of each of the `num_tx` transactions are assigned (this corresponds to the lines 7-15 in the algorithm pseudocode 4). 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**
     $\texttt{start}_i \leftarrow t_i.start, \ \texttt{end}_i \leftarrow t_i.end$
   - **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 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 19, 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 19: Example of transaction-all.csv

Anomalous Transaction Set

After the generation of regular transactions we perform an injection of transactions to produce anomalous scenarios. The injection is tailored depending on the specific kind of anomalous scenarios that we want to produce. In what follows we explain the injection process for the so far considered fraud pattern I.

To achieve this we inject transactions that violate the minimum *time-distance* constraint between consecutive transactions performed with the same card. Therefore, as we can see in Figure 24, if we consider a set of regular transactions for a certain card, where y_1 and y_2 are regular consecutive transactions, we will introduce an anomalous transaction a_{12} such that:

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

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

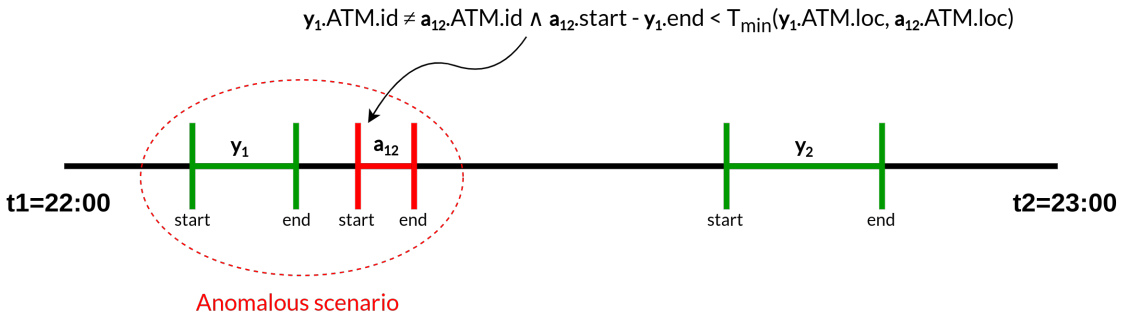


Figure 24: Creation of an anomalous scenario of the fraud pattern type I

Some assumptions related with the generation of anomalous transactions were made: (i) Overlapping of transactions is not possible; (ii) There are no two consecutive anomalous transactions. Both assumptions were made for simplicity in order to avoid creating other fraud patterns apart from the one considered so far (fraud pattern I). Note that, these assumptions could be omitted otherwise.

- **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. 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 the generation of anomalous transactions we do not allow the injection of two or more consecutive anomalous transactions. That is, an anomalous transaction can only be injected between two regular consecutive transactions. See Figure 25.

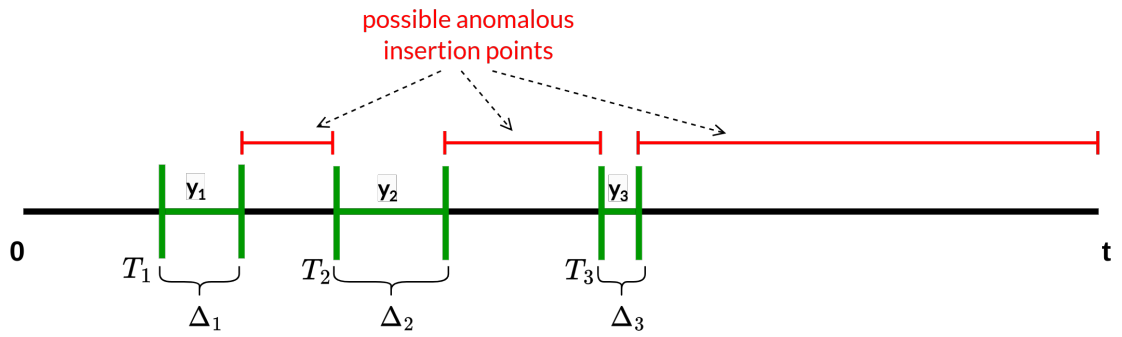


Figure 25: Considered possible injection points of anomalous transactions of fraud type I

ANOMALOUS_RATIO_1 $\in [0, 1]$ defines the ratio of anomalous transactions of the fraud pattern I over the total number of regular transactions num_tx for each of the cards. So that ANOMALOUS_RATIO_1 $*$ num_tx is the number of injected anomalous transactions for each of the cards.

In Algorithm 5 we describe at a high level the process of the generation of anomalous transactions for the fraud pattern I. This algorithm is run just after the generation of the regular transaction set, for each of the cards.

Algorithm 5 Injection of Anomalous Transactions of Fraud Pattern I

Precondition: This algorithm is run just after the creation of the regular transaction set for each of the cards. ATM_subset and its complement $\overline{\texttt{ATM\_subset}}$ are provided as parameters from the regular transaction generator, for each of the cards.

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

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

1. **Assignment of ATMs not belonging to the** ATM_subset: The anomalous transactions are linked to ATMs that are part of the complementary of the ATM_subset, the $\overline{\texttt{ATM\_subset}}$.

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 25, 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 respecting the needed time constraints**: In particular there are two time constraints to be satisfied:

    - Produce fraud pattern I 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 satisfying this two time constraints. To do it, we use the `ANOMALOUS_SPEED` (in km/h) as the assumed maximum speed at which the distance between the two ATMs of the regular and the anomalous transactions can be traveled. The time distance between the regular and the anomalous injected transaction is calculated based on this assumed maximum speed. So that a fraud pattern I is intended to be produced if the distance between the two ATMs is covered at a faster speed than the `ANOMALOUS_SPEED`, considered unfeasible. Finally `end` time is set as `end` = `start` + `ANOMALOUS_TX_DURATION`.

4. **Random transaction type**: Since the specific transaction *type* is not relevant for the production of the fraud pattern I, we assign it randomly (withdrawal, deposit, inquiry or transfer).

5. **Arbitrary amount**: Similarly, this is not a relevant property for the fraud pattern I, and therefore we just assign it an arbitrary amount such as $prev_i$.`transaction_amount`$*$ 2.

Transaction Stream Generator: `txGenerator.py`

We implemented the aforementioned generator of the synthetic transaction streams as a Python program `txGenerator.py`. On it we need to specify the value of the parameters needed to customize the generation of the stream of transactions. The customizable parameters and their description is provided in Table 9. 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>` introducing *outputFileName* as an argument to name the transaction stream 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.

Finally, a simplified version of this synthetic stream generator was developed in the Python program `txGenerator-simplified.py`. On it, the `ATM_subset` is built from a random selection of ATMs of the bank network, and not based on the distance to the residence location of the cardholder. This results in a faster generator, since it reduces the time complexity of the generation.

Parameter	Description
START_DATE	Start date (in date format: "YYYY-MM-DD")
NUM_DAYS	Number of days duration of the transaction stream generated
MAX_DISTANCE_SUBSET_THRESHOLD	Maximum allowed distance (in km) of the ATMs in the ATM subset to the client location residence
MAX_SIZE_ATM_SUBSET_RATIO	Ratio $\in [0, 1]$ to limit the maximum size of the ATM subset from which regular transactions of a card are linked to. So that: $\lvert\texttt{ATM\_subset}\rvert = \texttt{MAX\_SIZE\_ATM\_SUBSET\_RATIO} * \lvert\texttt{ATM}\rvert$
MAX_DURATION	Maximum time (in seconds) duration of a transaction
MEAN_DURATION	Mean duration (in seconds) duration of a transaction
STD_DURATION	Standard deviation (in seconds) duration of a transaction
REGULAR_SPEED	Assumed as the normal speed (in km/h) at which a client normally can travel the distance between two geographical points
ANOMALOUS_RATIO_1	Ratio $\in [0, 1]$ of anomalous transactions of the fraud pattern I over the total number of regular transactions for each of the cards.
ANOMALOUS_SPEED	Assumption on the maximum speed (in km/h) at which the distance between two geographical points can be traveled, for the generation of anomalous transactions
ANOMALOUS_TX_DURATION	Duration (in seconds) of an anomalous transaction

Table 9: Description of the customizable parameters for the transaction stream generation

6.4 Considered Datasets

Testing a bank system application like the $\mathsf{DP}_{\mathsf{ATM}}$ system implies deciding on different representative bank sizes and different stream configurations 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 propose two different bank databases: $\mathsf{GDB}_{\mathsf{A}}$ and $\mathsf{GDB}_{\mathsf{B}}$. $\mathsf{GDB}_{\mathsf{A}}$ simulates a small-sized bank database system, whereas $\mathsf{GDB}_{\mathsf{B}}$ intends to simulate a large bank database; such as the *Deutsche Bank Spain*[16]. In table 10 we give their details on its number of Cards and ATM entities.

Name	$\lvert\textbf{Card}\rvert$	$\lvert\textbf{ATM}\rvert$	$\lvert\textbf{ATM}_{\textbf{Internal}}\rvert$	$\lvert\textbf{ATM}_{\textbf{External}}\rvert$
$\mathsf{GDB}_{\mathsf{A}}$	2000	50	40	10
$\mathsf{GDB}_{\mathsf{B}}$	500000	1000	900	100

Table 10: Considered bank databases characteristics

6.4.1 Stream Configurations

Regarding the design of the synthetic generated stream of transactions, we did a brief investigation of some related works, with respect to the stream sizes as well as the ratio on the anomalous fraud transactions they utilize.

On [51] 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 [27] 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 [15] 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%.

Based on these references, for each considered bank database; $\mathsf{GDB_A}$ and $\mathsf{GDB_B}$, 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. 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 less than 1 transaction operation per day, in particular 0.666 transactions per day per cardholder on average (see table 5). This means that for each of the bank sizes being tested, in order to achieve the desired stream size, we needed to decrease/increase the size of the time interval to be simulated.

Table 11 presents summary of the details on the different transactions streams generated for each of the bank databases. The customizable parameter values that we used for the generation of these streams are detailed in table 12. All the generated streams were generated with those same common parameter values. The parameters that varied were: the `NUM_DAYS` to regulate the length of the transaction stream generated; and the `ANOMALOUS_RATIO_1`, to vary the ratio of anomalous transactions generated.

Name	**Bank DB**	**|Days|**	**Anomalous Ratio**	**Stream Size**	**|Regular Tx|**	**|Anomalous Tx|**
$\mathsf{GDB_A}$-30	$\mathsf{GDB_A}$	30	0.02 (2%)	39959	39508	451
$\mathsf{GDB_A}$-60	$\mathsf{GDB_A}$	60	0.02 (2%)	80744	79005	1739
$\mathsf{GDB_A}$-120	$\mathsf{GDB_A}$	120	0.02 (2%)	160750	157756	2994
$\mathsf{GDB_B}$-7	$\mathsf{GDB_B}$	7	0.03 (3%)	2428286	2401806	26480
$\mathsf{GDB_B}$-15	$\mathsf{GDB_B}$	15	0.03 (3%)	4856573	4805920	50653

Table 11: Summary of the different streams generated for each of the bank databases. For each stream, we indicate the time interval (in days) that it simulates, the anomalous ratio, its exact stream size (in the number of transactions), and from it, its number of regular and only anomalous transactions.

As a remark, note that for the $\mathsf{GDB_B}$ we utilized the simplified version of the generator, whereas for the $\mathsf{GDB_A}$ we utilized the original version.

Parameter	Value
START_DATE	2018-04-01
MAX_DISTANCE_SUBSET_THRESHOLD	70 (km)
MAX_SIZE_ATM_SUBSET_RATIO	0.2
MAX_DURATION	600 (s)
MEAN_DURATION	300 (s)
STD_DURATION	120 (s)
REGULAR_SPEED	50 (km/h)
ANOMALOUS_SPEED	500 (km/h)
ANOMALOUS_TX_DURATION	5 (s)

Table 12: Description of the values used for the customizable parameters of the transaction stream generator for the generated streams

6.5 Stream Ingestion

As it was already advanced in the description of the implementation of the *Source* Sr stage in 5, the way the stream of interactions is input into the $\mathsf{DP}_{\mathsf{ATM}}$ system is of paramount importance when evaluating the performance of a real-time system like ours.

In a real-case scenario, the interaction/transaction events coming from the network of ATMs of the bank would be typically received in a stream manner by a message queue of the $\mathsf{DP}_{\mathsf{ATM}}$ system. However, to test our proof of concept $\mathsf{DP}_{\mathsf{ATM}}$ system, we decided to simplify this process and do a file input read of the `csv` files containing our generated simulated synthetic transaction streams. 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.

For all the kinds of experiments we perform we want the reading of the input file to be the fastest possible, so to minimize the potential bottleneck derived from the I/O operation of reading a file. With this purpose, 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 20: `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 received 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.

The justification of the usage of this buffered and chunked file reading using the `encoding/csv` package with a *chunkSize* of 10^2 rows is provided with the upcoming results. On them 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.

Some references that we utilized include: [48, 45, 40, 49] are different blogs and tutorials where `Apache Arrow` is explained and where its usage with `Go` is also exemplified. [50] is an informal post where they experimentally showed some of the benefits of file reading in chunks.

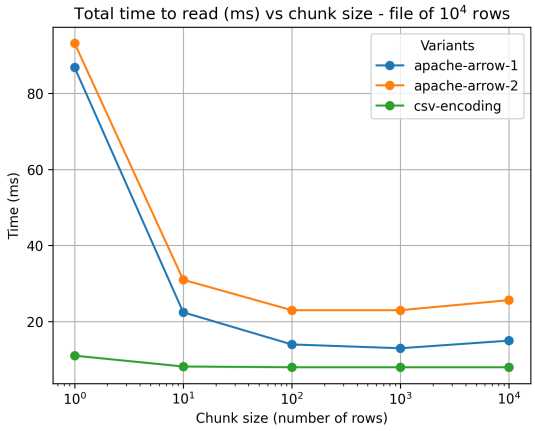
First the performance of the `encoding/csv` package is compared to the `apache/arrow` package. `encoding/csv` [18] is the package provided by the `Go` standard library to decode and encode data using `csv` values format. `apache/arrow` [8] is a `Go` package from the `Arrow` platform [9] that allows reading `csv` files in chunks of n rows, called *records*. An inconvenient is that `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.

All the compared approaches do the reading by chunks in a worker subroutine. The chunks are then provided to the main process through an internal communication channel. This intends to simulate a real implementation of the *Source* Sr stage.

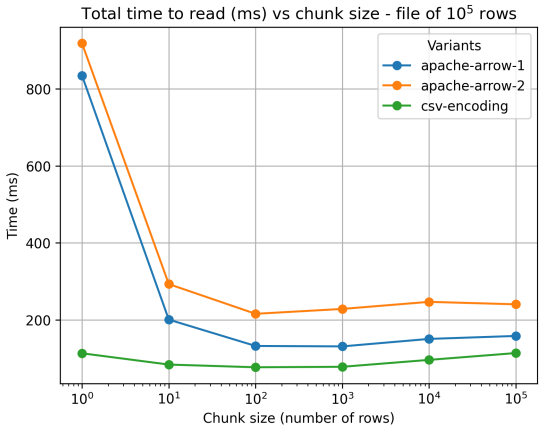
- `apache/arrow-1`: Reading done with `apache/arrow` package. The worker performs the reading of each field of the `csv` with its corresponding data type. After reading a chunk, it is then transposed back to obtain the `Edge` type rows (as the library optimizes saving the `csv` by columns when read), and the chunk of `Edge` rows given to the main process.
- `apache/arrow-2`: Reading done with `apache/arrow` package. The worker reads each field as `string` data type. After reading a chunk, it is transposed bank to row form and sent to the main process. The conversion to the corresponding `Edge` types is performed in the main process.
- `csv/encoding`: Reading done using the `encoding/csv` package. Row by row reading by the worker. When a chunk of rows is formed it passes it to the main process and the rows are then converted to `Edge`'s.

To perform these experiments we generated `csv` stream files of toy interactions of different sizes: 10^4, 10^5 and 10^6 number of interactions (rows). For each of the sizes we compared the time it took to read the full file to each of the variants, also testing with 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, i.e. all at once). Each of the experiments performed were run a total of 20 times to obtain stable measurements. The results can be seen in Figure 26. In all of the cases, the fastest approach is the variant

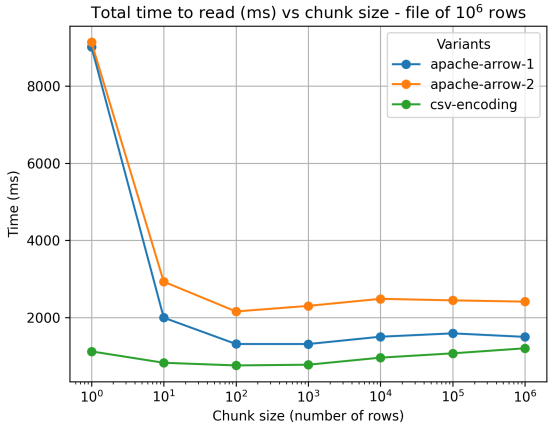
using `encoding/csv` with chunk size *chunkSize* of 10^2 rows.



Test for `csv` file of size 10^4 rows



Test for `csv` file of size 10^5 rows

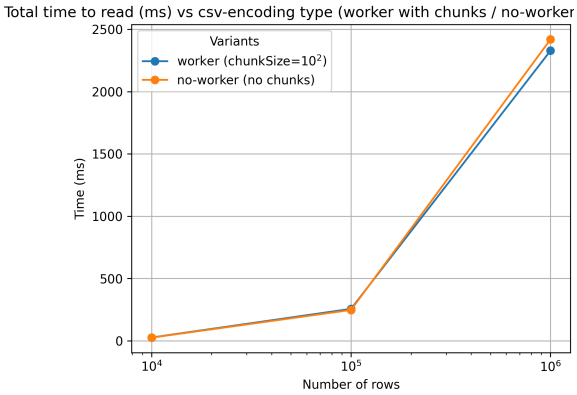


Test for `csv` file of size 10^6 rows

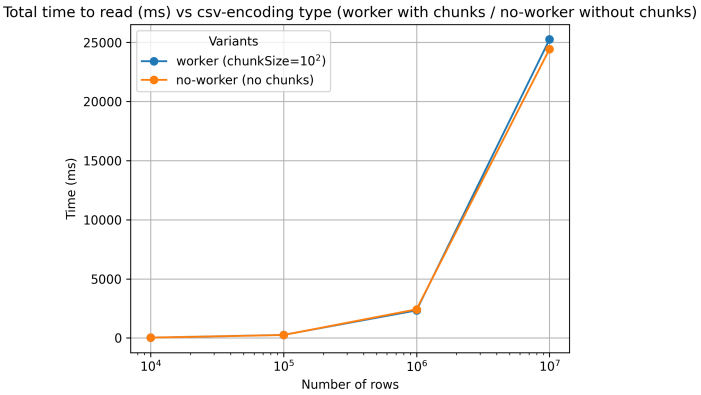
Figure 26: Comparison of the `apache/arrow-1`, `apache/arrow-2` and `encoding/csv` variants for reading different `csv` file sizes, using different chunk sizes.

Once we decided to use the approach using the `encoding/csv` package, we performed an additional experiment in order to see if it was actually worthy to do the *background* reading of the input with the worker subprocess `goroutine`. To see this we performed some experiments in which we compared the variant with worker and chunk size of 10^2 with respect to the one without worker and without chunk reading. Again we compared the time it took to read `csv` interaction files of different sizes : from 10^4 up to 10^6 and 10^7 rows/interactions. Each experiment was done 20 times to obtain stable measurements.

As it can be seen in Figure 27, the differences are insignificant. Up to 10^6 rows the variant with worker and chunk reading seems to be slightly better, although for the experiment with 10^7 rows is the other way around.



Test for `csv` file of size up to 10^6 rows



Test for `csv` file of size up to 10^7 rows

Figure 27: Comparison of the worker and no worker variants using `encoding/csv` for reading different `csv` file sizes.

All these experiments were run with 1 core and 1024MB of RAM at the nodes of the RDLab-UPC cluster.

7 Analysis of Experimental Results

In what follows we introduce the following notation to refer to the different experiments performed. Let $\Sigma = (g, s(k,p), f, c)$ represent the execution of an experiment where:

- g is a datagraph;
- $s(k,p)$ represents a stream of transactions of length k and percentage p of anomalous transactions;
- f is the number of filters;
- c is the number of cores.

We perform experiments for:

- $g = \mathsf{GDB_A}$ and $\mathsf{GDB_B}$;
- $s(k,p)$:
  - With $g = \mathsf{GDB_A}$ for $k = 30, 60, 120$ and $p = 0.02$.
  - With $g = \mathsf{GDB_B}$ for $k = 7, 15$ and $p = 0.03$.
- f:
  - With $g = \mathsf{GDB_A}$ for $f = 1, 2, 5, 10, 20, 40, 100, 200, 500, 1000, 2000$.
  - With $g = \mathsf{GDB_B}$ for $f = 1, 5, 10, 100, 250, 500, 1000, 2000, 5000, 10000$.
- $c = 1, 2, 4, 8, 16$

For instance, $\Sigma(\mathsf{GDB_A}, s(30, 0.02), f, c)$ refers to the experiment performed with $g = \mathsf{GDB_A}$, $k = 30$ days stream length and $p = 0.02$ anomalous ratio, performed for all the filters f combinations possible for $g = \mathsf{GDB_A}$, and for all cores c combinations.

E1: Evaluation in a High-Load Stress Scenario

The evaluation of the $\mathsf{DP_{ATM}}$ in this scenario was done for the two different bank databases; $\mathsf{GDB_A}$ and $\mathsf{GDB_B}$. For each simulated bank database system we provided the transactions streams that we generated. T he details on the generated datasets are explained on 6.4. Let the Tables 13 and 14 serve as a summary.

Name	**|Card|**	**|ATM|**	**$\|\mathbf{ATM_{Internal}}\|$**	**$\|\mathbf{ATM_{External}}\|$**
$\mathsf{GDB_A}$	2000	50	40	10
$\mathsf{GDB_B}$	500000	1000	900	100

Table 13: Bank databases characteristics

Name	Bank DB	|Days|	Anomalous Ratio	Stream Size	|Regular Tx|	|Anomalous Tx|
GDB_A-30	GDB_A	30	0.02 (2%)	39959	39508	451
GDB_A-60	GDB_A	60	0.02 (2%)	80744	79005	1739
GDB_A-120	GDB_A	120	0.02 (2%)	160750	157756	2994
GDB_B-7	GDB_B	7	0.03 (3%)	2428286	2401806	26480
GDB_B-15	GDB_B	15	0.03 (3%)	4856573	4805920	50653

Table 14: Summary of the different streams generated for each of the bank databases. For each stream, we indicate the time interval (in days) that it simulates, the anomalous ratio, its exact stream size (in the number of transactions), and from it, its number of regular and only anomalous transactions.

For each bank DP_{ATM} system to be tested, we considered different DP_{ATM} configurations regarding the number of filters with which we construct the DP pipeline. This is controlled by setting up the system parameter *maxFilterSize*, which determines the maximum number of cards each *Filter* stage tracks. Table 15 represents the different *maxFilterSize* configurations tested for each of the bank systems.

Each of these DP_{ATM} configurations were run for different resources variations in terms of the dedicated number of cores, in particular for c =1, 2, 4, 8 and 16 cores. For each particular number of cores setting the DP_{ATM} configurations were compared against a sequential baseline version of our system (cores = 1). This sequential version is referred in the following results as both the *sequential/baseline* or as the `0-filter` version. It is a simple program version of the DP_{ATM} that executes the same algorithm but without a DP, that is, only with a main process that process the transactions one after the other tracking the activity of all the cards at once. It also implements the same stream ingestion method as the one referred in 6.5. However, it is different from the DP_{ATM} with one single *Filter* stage in the sense that, this last contains a proper *Filter* F stage with its respective *Filter Worker* FW substage, as well as a *Source* Sr and a *Sink* Sk stage.

|Cards| per *Filter* (*maxFilterSize*)	***|Filter|***
2000	1
1000	2
400	5
200	10
100	20
50	40
20	100
10	200
4	500
2	1000
1	2000

$\mathsf{GDB_A}$

|Cards| per *Filter* (*maxFilterSize*)	***|Filter|***
500000	1
100000	5
50000	10
5000	100
2000	250
1000	500
500	1000
250	2000
100	5000
50	10000
10	50000

$\mathsf{GDB_B}$

Table 15: For $\mathsf{GDB_A}$ and $\mathsf{GDB_B}$, the different $\mathsf{DP_{ATM}}$ configurations combinations for the *maxFilterSize* parameter, including the derived number of *Filter* stages of the $\mathsf{DP_{ATM}}$.

Each job was run ten times in the case of the experiments of the $\mathsf{GDB_A}$ and one in the case of the $\mathsf{GDB_B}$ experiments, due to time constraints and its longer execution times. We limited to `16`GB of RAM the amount of memory used for each experiment. Note that in the case of the $\mathsf{GDB_B}$, the tests with 50000 filters could not be performed due to this memory limitation.

$\mathsf{DP_{ATM}}$ Comparison to Sequential Baseline

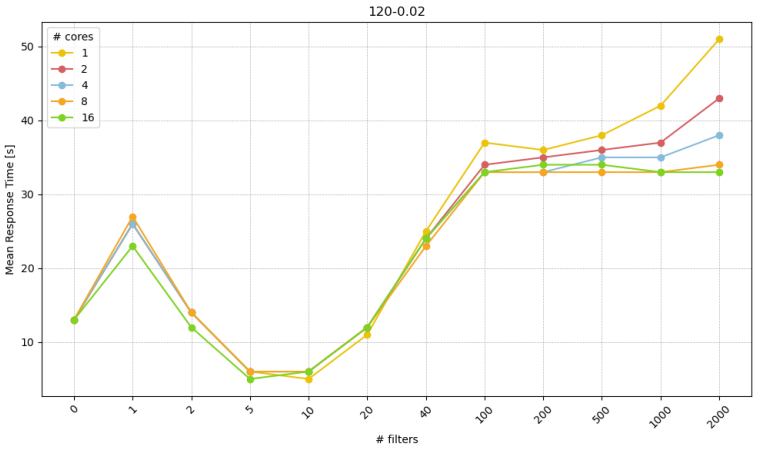
First we provide and study the results to compare our different $\mathsf{DP_{ATM}}$ configurations (with the different number of filters variations) with respect to the sequential version of the program, the `0-filter` version.

In Figure 28 we can see the comparison of different metrics results for the experiment $\Sigma(\mathsf{GDB_A}, s(120, 0.02), f, c)$. We can see that the performance of the sequential version is, in general, the worst regarding all the metrics except for the Mean Response Time (`MRT`), where the `0-filter` exhibits a lower `MRT` than many configurations of the $\mathsf{DP_{ATM}}$, no matter the number of cores (28a).

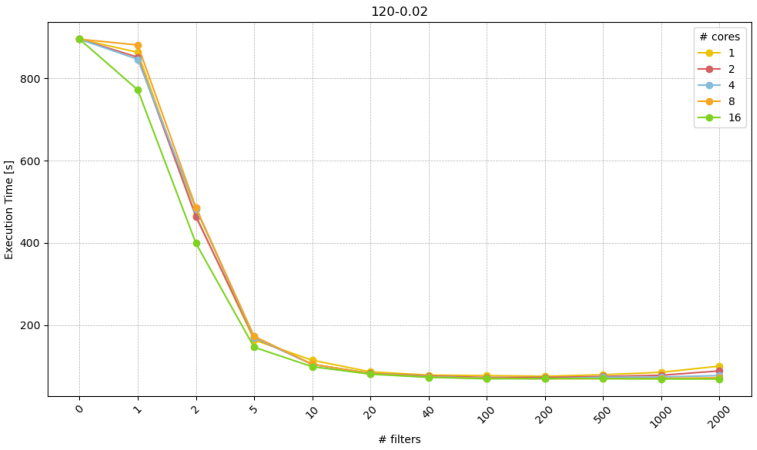
In this sense, the sequential version achieves a lower `MRT` for any configuration with a number of filters above 40. Only the variations with a number of filters in the range 5-20 are able to outperform the sequential version on this metric, showing a `MRT` around 5-6 seconds in the 5 and 10 filter cases, for all the number of cores tested. To get more insights on this, we show in Figure 29 the Response Time (`RT`) trace of the different configurations

run with 16 cores. On it, as mentioned, we can see that only the configurations with a number of filters in the range 5-20 tend to have a RT below the sequential version for all of the checks. The 1-filter version is always above. However, an interesting phenomena that we see is that for the versions with 40 and more filters the RT presents a linear growing shape, whereas for the other versions the RT tends to be constant for all the performed checks. These *large* number of filters versions even present a lower RT up to a certain number of checks. Later we will come again to this discussion, so to analyze better why this could be the cause of this behavior.

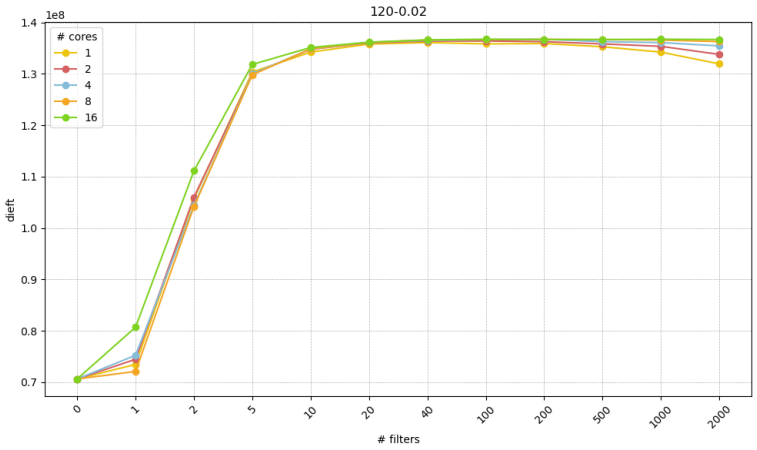
Nevertheless, as expected, the sequential version underperforms for the other metrics such as the Execution Time ET (28b), the throughput T (28d) and the dief@t (28c). In Figure 30 we show the comparison of all these metrics together with a radar plot for the experiment $\Sigma(\mathsf{GDB_A}, s(120, 0.02), f, 16)$. A similar behavior can be observed in the experiments done for $\mathsf{GDB_B}$. In Figure 31, we show the radar plot of the $\Sigma(\mathsf{GDB_B}, s(7, 0.03), f, 16)$ experiment. On it, again, only the versions with a number of filters in the range 5-20 outperform the sequential version in the MRT. However, for the rest of the metrics we can observe that, in general, almost all $\mathsf{DP_{ATM}}$ configuration exhibits a better performance than the sequential version.



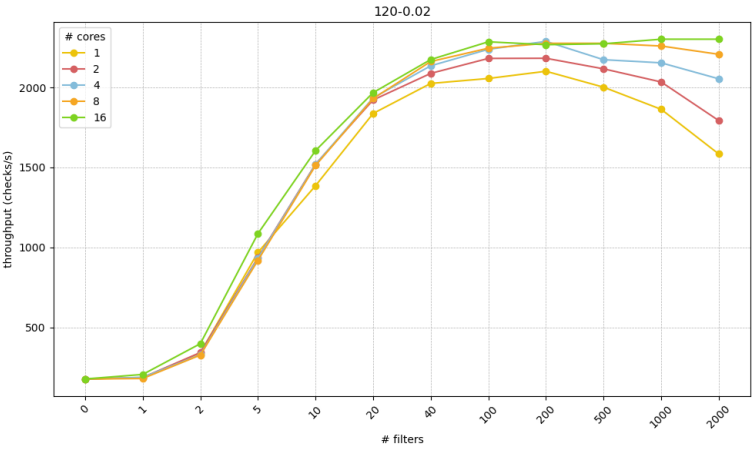
(a) Mean Response Time `MRT` (in seconds)



(b) Execution Time `ET` (in seconds)



(c) `dief@t`



(d) Throughput `T` (checks/s)

Figure 28: Metrics comparison for experiment $\Sigma(\mathsf{GDB}_\mathsf{A}, s(120, 0.02), f, c)$. Each color represents the experiment run with a different number of cores. (a) `MRT`, (b) `ET`, (c) `dief@t`, (d) `T`.

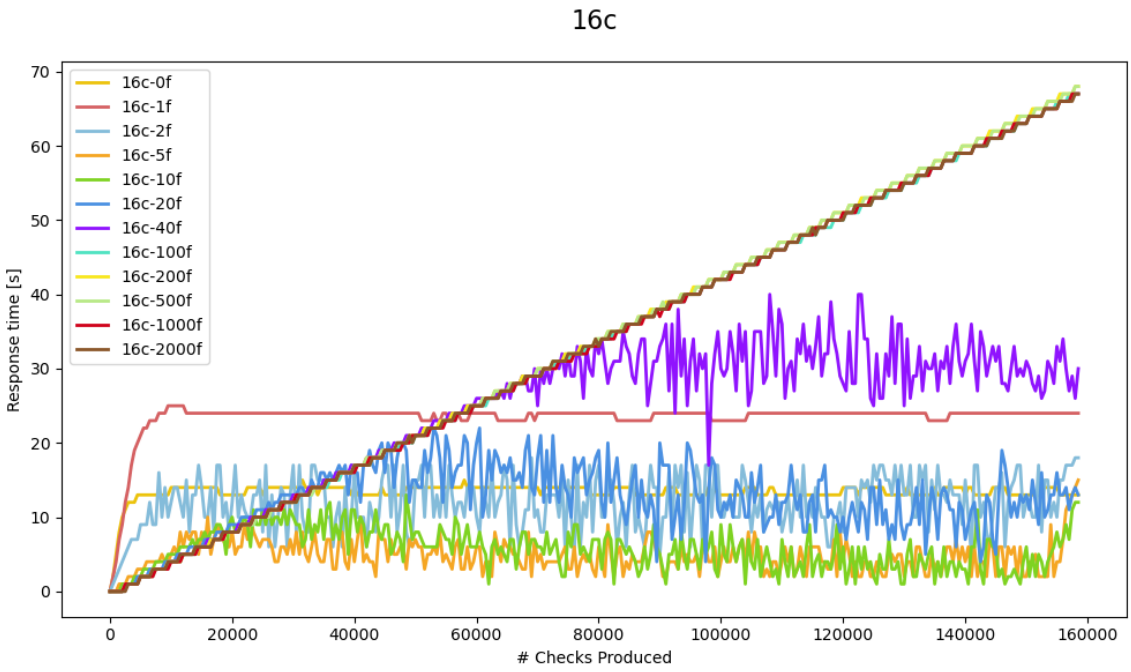


Figure 29: Response Time (RT) trace for the $\Sigma(\mathsf{GDB}_{\mathsf{A}}, s(120, 0.02), f, 16)$ test. Horizontal axis shows the *check* number and the vertical axis shows the RT (in seconds) for that check.

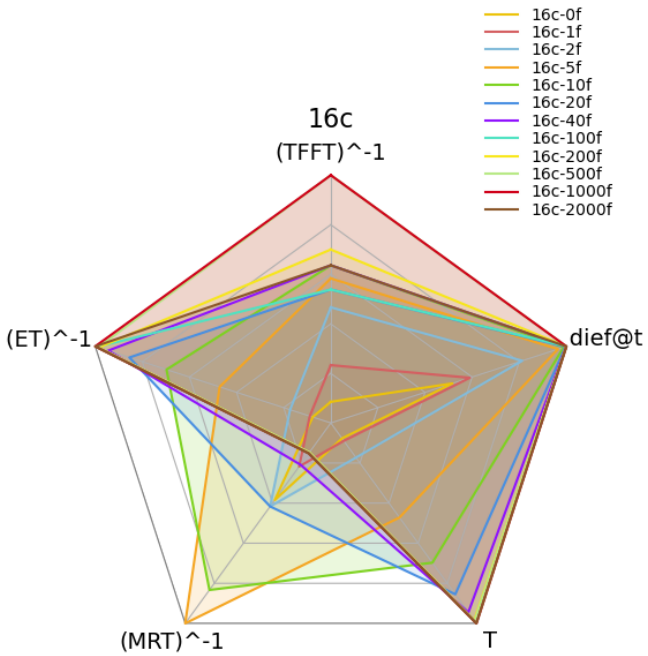


Figure 30: Metrics radar plot comparison for the $\Sigma(\mathsf{GDB}_{\mathsf{A}}, s(120, 0.02), f, 16)$ experiment. Each axis represents a metric: $\texttt{MRT}^{-1}$, T (Throughput), dief@t, $\texttt{TFFT}^{-1}$ and $\texttt{ET}^{-1}$. Better performance for a metric is achieved whenever the value is closer to the outer boundary of the corresponding axis.

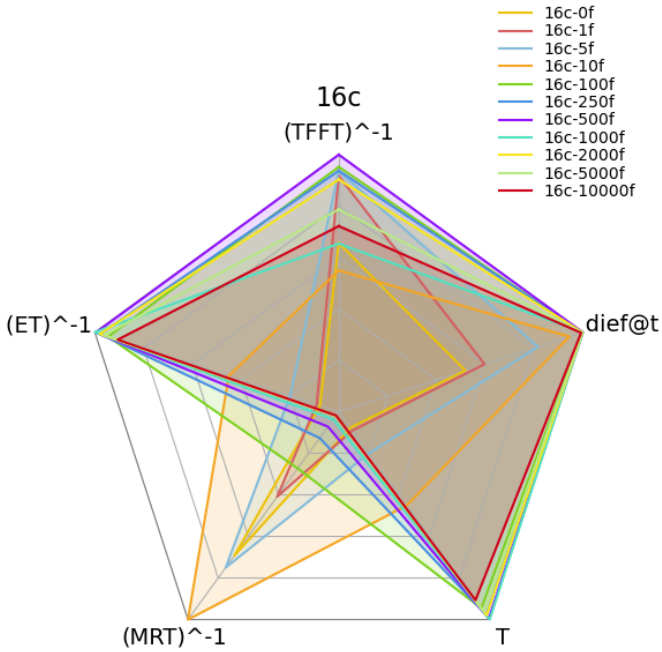


Figure 31: Metrics radar plot comparison for the $\Sigma(\mathsf{GDB}_\mathsf{B}, s(7, 0.03), f, 16)$ experiment. Each axis represents a metric: $\texttt{MRT}^{-1}$, `T` (Throughput), `dief@t`, $\texttt{TFFT}^{-1}$ and $\texttt{ET}^{-1}$. Better performance for a metric is achieved whenever the value is closer to the outer boundary of the corresponding axis.

Analysis of the Number of Filters Configuration

Regarding the configuration of the DP_ATM system, we are interested in analyzing which is the best configuration in terms of the number of filters for each the GDB_A and GDB_B, given a certain number of cores resource limitation.

For GDB_A, as already mentioned for the $\Sigma(\mathsf{GDB}_\mathsf{A}, s(120, 0.02), f, c)$ test, regarding the `MRT` the configurations using 5-10 number of filters exhibit the best `MRT`, of around 5-6 seconds, worsening for the configurations with a large number of filters (see Figure 28a). Indeed they show a constant and low response time in comparison with the ones with large number of filters; see Figure 28b.

Nonetheless, these configurations do not perform that well in the case of the `ET` or the `dief@t`, where a larger number of filters seems to be a better option, being able to consume the input stream faster and showing a better continuous behavior in terms of `dief@t`. However, it is worthy to note that the behavior tends to degrade for all the metrics in the case of having a low number of cores. We can see this clearly for the case of the runs with `1` core. It can be seen that, in this case, a large number of filters could be causing an overhead and therefore producing this degradation of the behavior for the `ET` and the `dief@t`. We can observe this in all the metrics figures of Figure 28, where specially for the `1` core variations, we can appreciate the degradation for the large number of filters configurations (in the rightmost part of each of the plots). As expected, more cores help to improve the behavior, especially for the variants with large number of filters.

The analysis on the number of filters configuration is quite similar in the case of the

experiments done for GDB_B. For instance, in Figure 32 we show the metrics radar for $\Sigma(\mathsf{GDB}_\mathsf{B}, s(15, 0.03), f, 8)$, where we can also see that the configurations that achieve the best `MRT` are the ones with a number of filters in the range 5-10, whereas they are the ones that underperform in the `ET` and `dief@t` metrics. This can also be seen in the results trace of this same test on Figure 33, where the continuous behavior for these filter configurations underperforms the continuous behavior for configurations with higher number of filters. However, they have a better continuous behavior than the sequential or the 1 filter configurations.

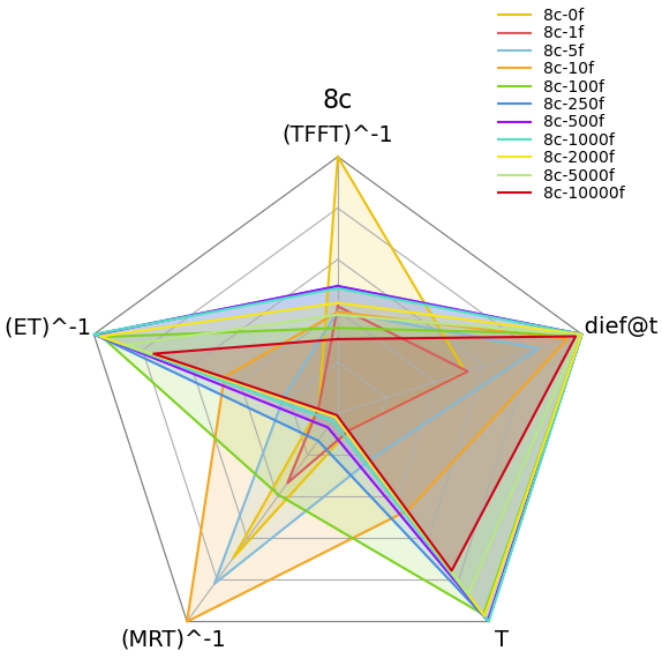


Figure 32: Metrics radar plot comparison for $\Sigma(\mathsf{GDB}_\mathsf{B}, s(15, 0.03), f, 8)$. Each axis represents a metric: $\texttt{MRT}^{-1}$, `T` (Throughput), `dief@t`, $\texttt{TFFT}^{-1}$ and $\texttt{ET}^{-1}$. Better performance for a metric is achieved whenever the value is closer to the outer boundary of the corresponding axis.

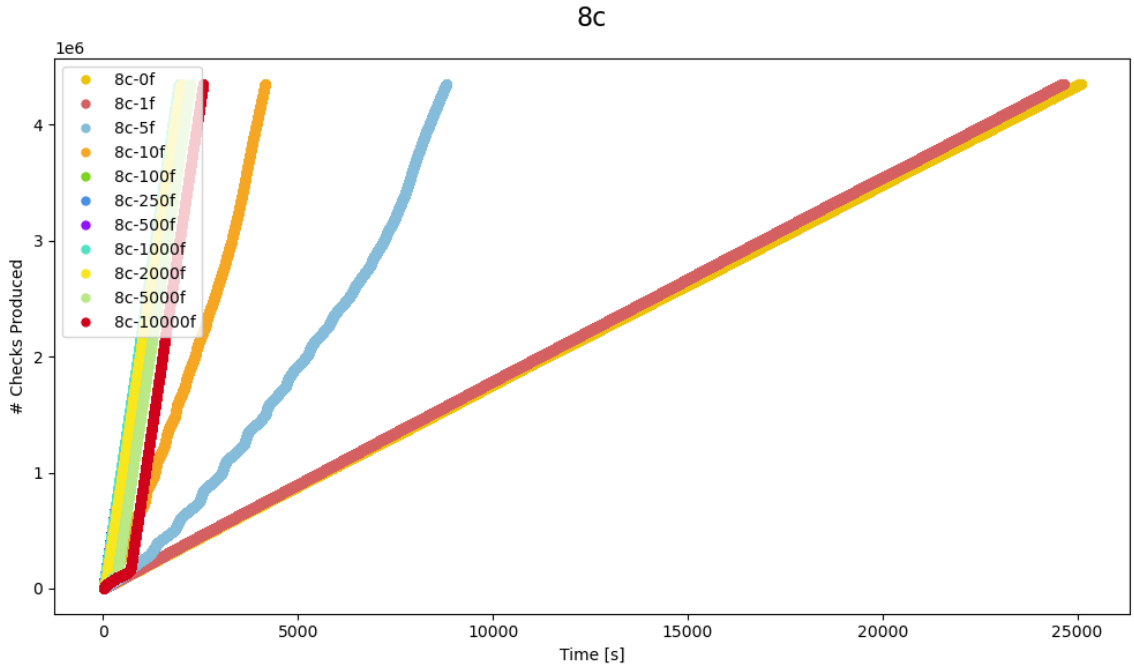


Figure 33: Check results trace for $\Sigma(\mathsf{GDB}_\mathsf{B}, s(15, 0.03), f, 8)$. Pairs (result, time(s)). Vertical axis shows the *check* number and horizontal axis shows the time in seconds at which that check is produced.

Another aspect in which the configurations in the range of 5-10 number of filters outperform, apart from achieving the lowest `MRT` is in achieving it through a constant `RT` along all the execution. Figure 34 shows the response time trace for the $\Sigma(\mathsf{GDB}_\mathsf{B}, s(15, 0.03), f, 8)$. On it, the 5-filter variant maintains a `RT` around 11 seconds and the 10-filter around 9 seconds, in comparison to other larger number of filters variations that end up accumulating more than 100s of `RT`.

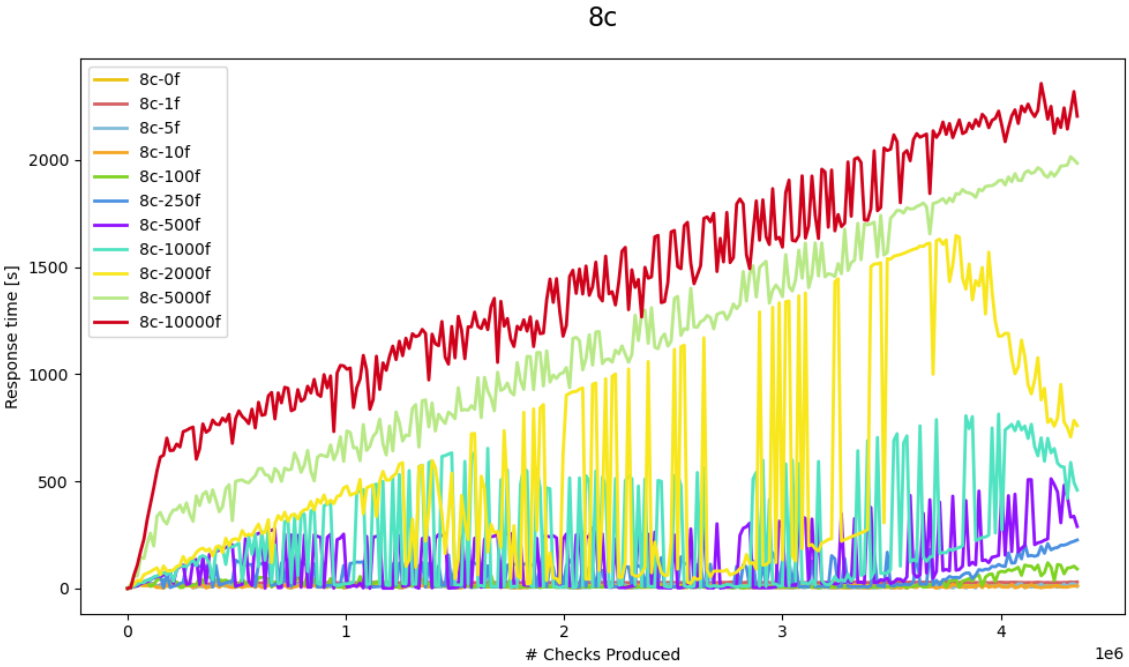
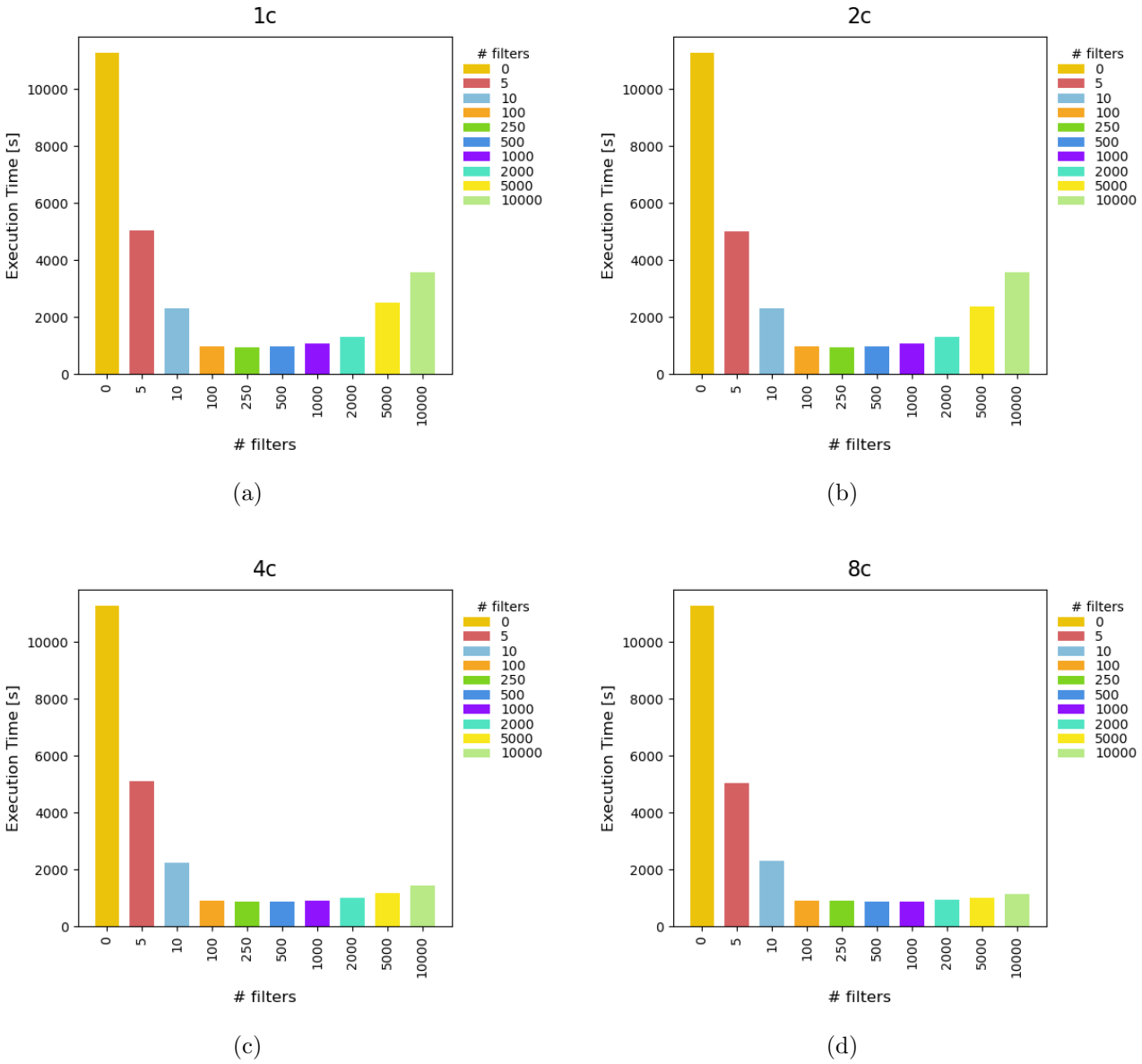


Figure 34: Response Time (RT) trace for $\Sigma(\mathsf{GDB}_\mathsf{B}, s(15, 0.03), f, 8)$. Horizontal axis shows the *check* number and the vertical axis shows the RT (in seconds) for that check.

In addition, the degradation of the behavior of the configurations with a high number of filters for a low number of cores executions is a phenomena that we can also see in the GDB_B experiments. For instance we can see the degradation of the ET in Figure 35 in the case of the test $\Sigma(\mathsf{GDB}_\mathsf{B}, s(7, 0.03), f, c)$. On it, we observe how the ET deteriorates for the configurations with a high number of filters (2000, 5000 and 10000) whenever we reduce the number of cores. This can be due to: (i) an overhead on the number of `goroutines` utilized and the overhead in the communication of the pipeline that this might be causing; (ii) the overhead on the communication to the Neo4j graph database instance through those many parallel sessions, remember that we have one parallel session per filter stage; (iii) a bottleneck at the *Sink* Sk stage, caused by its current file writing of the results. However, note that in a real implementation of the system the results emission would be done in another manner, such as the emission of the alerts through the network to the cardholder. In any case, further investigation and potential improvements to address this issue will be considered as future work.



(a) (b) (c) (d)

Figure 35: ET for $\Sigma(\mathsf{GDB}_\mathsf{B}, s(7, 0.03), f, c)$. (a) Run with $c = 1$ core. (b) Run with $c = 2$ cores. (c) Run with $c = 4$ cores. (d) Run with $c = 8$ cores.

Finally, since we consider MRT one of the most important metrics for assessing the suitability of our DP_ATM as a real-time system for the rapid detection of card-ATM fraud, we show Tables 16 and 17 on which we show the MRT values in seconds for the experiments performed on the GDB_A and GDB_B largest tested streams, $k = 120$ and $k = 15$, respectively. On them we can show that, in the case of the $\Sigma(\mathsf{GDB}_\mathsf{A}, s(120, 0.02), f, c)$ the best filter (5 and 10 filters) configuration has a MRT around 5-6 seconds for all the tested number of cores variation. For the $\Sigma(\mathsf{GDB}_\mathsf{B}, s(15, 0.03), f, c)$, the best filter configurations in this metric, again the 5 and 10 filter configurations, achieve a MRT around 10 seconds.

MRT (s) for $\mathsf{GDB_A}$-120					
Number of Cores					
# Filters	**1**	**2**	**4**	**8**	**16**
0	13	13	13	13	13
1	26	26	26	27	23
2	14	14	14	14	12
5	6	6	6	6	5
10	5	6	6	6	6
20	11	12	12	12	12
40	25	24	24	23	24
100	37	34	33	33	33
200	36	35	33	33	34
500	38	36	35	33	34
1000	42	37	35	33	33
2000	51	43	38	34	33

Table 16: Mean Response Time (`MRT`) in seconds for the $\Sigma(\mathsf{GDB_A}, s(120, 0.02), f, c)$ experiment.

MRT (s) for $\mathsf{GDB_B}$-15					
Number of Cores					
# Filters	**1**	**2**	**4**	**8**	**16**
0	13	13	13	13	13
5	11	10	11	11	9
10	11	9	10	9	7
100	33	40	29	23	37
250	124	80	91	69	90
500	221	161	134	136	138
1000	502	301	267	256	263
2000	937	702	618	562	554
5000	2830	1732	1422	1124	1060
10000	4957	2535	2025	1410	1249

Table 17: Mean Response Time (`MRT`) in seconds for the $\Sigma(\mathsf{GDB_B}, s(15, 0.03), f, c)$ experiment.

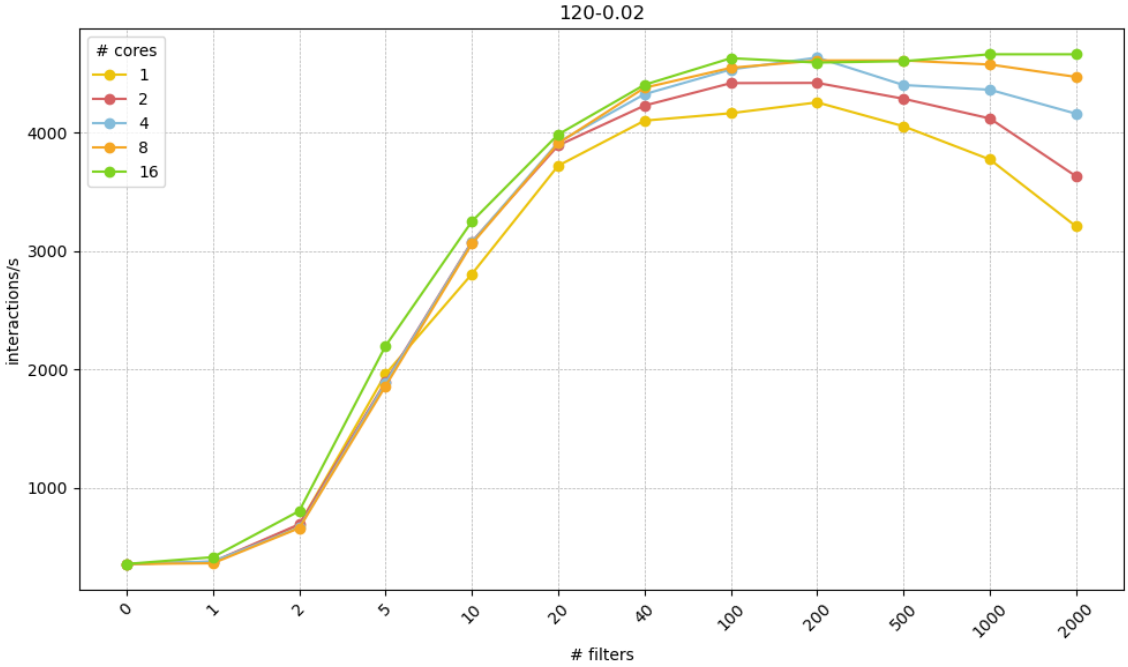


Figure 36: Processed `interactions/s` on the experiment $\Sigma(\mathsf{GDB}_\mathsf{A}, s(120, 0.02), f, c)$. Horizontal axis shows the variation in the number of filters f. Vertical axis shows the number of processed `interactions/s`. Each color represents the run with a different number of cores c.

To sum up, we can say that there is a trade-off between `MRT` and the other metrics. The DP_ATM configurations that offer the best balance across all metrics are those with 5 to 10 filters. We reason that this is the case since, as we already discussed in 6.1.2, we consider that a real-time system in this high-load stress scenario needs to show a good performance in various aspects. First, have a low and non-increasing - if possible - response time, so to allow the user / bank system be able to prevent potential the current and future frauds as fast as possible. In this sense, these configurations achieve reasonable low `MRT` times, in all the cases lower than 11 seconds. Secondly, they also have a reasonable good continuous behavior and have a good performance in terms of the speed to process the stream input. For instance, in $\Sigma(\mathsf{GDB}_\mathsf{A}, s(120, 0.02), f, c)$ experiment, we are able to achieve a rate of almost 5,000 `interactions/s` (2,500 transactions/s) processed (see the Figure 36). Therefore, we conclude that these DP_ATM configurations achieve a good balance across all these aspects.

8 Conclusions

In this work we address the problem of continuous query evaluation on an evolving graph database. To do so we decompose the datagraph into two well define subgraphs namely the *stable* subgraph and the *volatile* subgraph and use a stream processing approach to query these two subgraphs simultaneously. We use the dynamic pipeline computational model to process the stream data since its parallel concurrent nature has proven its suitability for developing real-time systems that emit results as they are computed, in a progressive way. And our preliminary experimental results show that this is also a suitable practical model to solve our problem of study. We provide a working open source continuous query engine, the $\mathsf{DP}_{\mathsf{ATM}}$ (available here [14]), implemented in the `Go` programming language, to detect abnormal or suspicious ATM transactions. Up to our knowledge, most of the work addressing this topic provide a delayed detection based on predictions given by ML systems while with our engine the detection is almost immediate up the datagraphs and datastreams that we were able to experimentally study. Due to the sensitive and confidential nature of banking data and transactions, there are no repositories offering this type of datasets for empirical studies. So, to be able to test our engine we have created synthetic repositories for this purpose. Therefore, as a by product of our engine we provide the developers community with a program to generate synthetic banking data under demand under the standard Neo4j graph data base model (available here [14]).

To test the $\mathsf{DP}_{\mathsf{ATM}}$ engine we considered two different sizes of data graphs, simulating the sizes of both a small real bank and a medium-size real bank (unfortunately due to time and hardware restrictions we were not able to test the engine for data graphs of similar size to a big real bank). For each data graph we consider two different scenarios: (i) a real time scenario in which the data stream simulates a real time situation in which every transaction in the stream of transactions has associated its exact time of occurrence and (ii) a highly stressed scenario in which transactions are considered to be continuous and the only difference between the occurrence time of two consecutive transactions is the time taken by the engine to read the two items consecutively. The second kind of experiment was proposed as a stress test for the system after observing that it worked really well with the real time experiments proposed at first.

All the tests were done considering the sequential framework as baseline: settings using different number of cores, filters, input stream sizes, percentages of abnormal transactions, etc., and comparing the $\mathsf{DP}_{\mathsf{ATM}}$ engine behavior against a sequential framework to solve the same problem.

Among all the considered metrics the response time `RT` and mean response time `MRT`, the execution time `ET` and the `dief@t` are the most relevant ones and in particular, looking at the `MRT`, it is interesting to observe that it has a relative low value in general and it does not seem to grow but to remain constant.

The main conclusion that we extract after the results of our experimental work and the observation of the applied metrics is that the $\mathsf{DP}_{\mathsf{ATM}}$ engine a real-time system capable of almost immediately inform whether an anomalous fraud situation entered the system.
As general observations derived from our experimentation we consider also worth mentioning that:

- The $\mathsf{DP}_{\mathsf{ATM}}$ beats sequential baseline (i.e. it obtains better results in general) for

almost all metrics – only for the `MRT` under some configurations of number of filters is it worse.

- The configurations with a small number of filters (between 5 and 10) are apparently the best regarding average response time while regarding other metrics, such as execution time or `dief@t`, configurations with more filters are better.
- Over all the run tests the $\mathsf{DP}_{\mathsf{ATM}}$ achieved a 100% of accuracy, meaning that it was able to detect all the fraudulent transactions. This is in contrast with other techniques, such as data mining, used in current applications that provide solutions with a much lower percentage of accuracy.

Although it was not possible to perform experiments with data graphs of sizes similar to those of big banks the experiments performed under the stressed scenario in both kinds of data graphs can be observed of simulations of a much larger banking since a much larger flow of transactions is being processed than that of that small bank. For example, in the $\Sigma(\mathsf{GDB}_{\mathsf{A}}, s(120, 0.02), f, c)$ test, the system processes almost 5,000 interactions (2,500 transactions per second) 36 which implies 3,600,000 per day, a number consequent with the number of transactions of a really big bank.

Last but not least, we claim that with our engine, in presence of some weird finding in an ATM transaction, banks have a tool able to either ask card holders for authorizations or to take any other fraud preventing action at real-time, instead of the frequent and annoying classical treatment of the problem that works by consulting log files because of the complain of customers when they themselves detect some weird movement in their accounts.

To go further with this work it would be interesting to investigate more on why the average response time grows so much for the combinations with more filters (it might be due to a bottleneck in the sink of the system, among other possible reasons), to make experiments with larger banking databases since the settings to generate the data graphs are already implemented, to include the study of more types of ATM frauds or even other kind of frauds (such as Point-Of-Sale frauds) and to study the problem of window management so that filters could be more dynamic and therefore could stop, at some point of their execution and under some specific policies, "tracking" the activity of a card whenever it is decided that too much time has passed since its last operation.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases*. Vol. 8. Addison-Wesley Reading, 1995.

[2] Maribel Acosta, Maria-Esther Vidal, and York Sure-Vetter. "Diefficiency Metrics: Measuring the Continuous Efficiency of Query Processing Approaches". In: Oct. 2017, pp. 3–19. ISBN: 978-3-319-68203-7. DOI: 10.1007/978-3-319-68204-4_1.

[3] Mohiuddin Ahmed, Abdun Naser Mahmood, and Md Rafiqul Islam. "A survey of anomaly detection techniques in financial domain". In: *Future Generation Computer Systems* 55 (2016), pp. 278–288.

[4] American Express. *Lost and Stolen Card Fraud*. Accessed: 2025-01-05. 2025. URL: https://www.americanexpress.com.kw/en-kw/fraud-protection-center/lost-and-stolen-card-fraud/.

[5] Renzo Angles. "The Property Graph Database Model". In: *CEUR Workshop Proceedings, CEUR-WS.org (AMW2018)*. Vol. 2100. 2018. URL: https://ceur-ws.org/Vol-2100/paper26.pdf.

[6] Renzo Angles and Claudio Gutierrez. "Survey of graph database models". In: *ACM Computing Surveys (CSUR)* 40.1 (2008), pp. 1–39.

[7] Renzo Angles et al. "Foundations of modern query languages for graph databases". In: *ACM Computing Surveys (CSUR)* 50.5 (2017), pp. 1–40.

[8] Apache. *Apache Arrow Go Package*. Accessed: 2025-01-20. 2025. URL: https://pkg.go.dev/github.com/apache/arrow/go/arrow/csv.

[9] Apache. *Arrow Platform*. Accessed: 2025-01-20. 2025. URL: https://pkg.go.dev/github.com/apache/arrow/go/arrow.

[10] Shivnath Babu and Jennifer Widom. "Continuous queries over data streams". In: *ACM Sigmod Record* 30.3 (2001), pp. 109–120.

[11] Shivnath Babu and Jennifer Widom. "Continuous queries over data streams". In: *ACM Sigmod Record* 30.3 (2001), pp. 109–120.

[12] Edward A Bender. "Partitions of multisets". In: *Discrete Mathematics* 9.4 (1974), pp. 301–311.

[13] Daniel Benedí García. "Maintaining the minimum spanning forest of fully dynamic graphs on the dynamic pipeline computational pattern". PhD thesis. UPC, Facultat d'Informàtica de Barcelona, Departament de Ciències de la Computació, June 2024. URL: http://hdl.handle.net/2117/417505.

[14] Fernando Martín Canfrán. *ATM-DP Implementation on Github*. https://github.com/FCanfran/ATM-DP. Accessed: 2025-01-20. 2025.

[15] Alejandro Correa Bahnsen et al. "Feature Engineering Strategies for Credit Card Fraud Detection". In: *Expert Systems with Applications* 51 (Jan. 2016). DOI: 10.1016/j.eswa.2015.12.030.

[16] Deutsche Bank España. *Quiénes somos - Deutsche Bank en España*. Accessed: 2025-01-21. 2024. URL: https://country.db.com/spain/quienes-somos/en-espana/.

[17] Go Runtime. *GOMAXPROCS Environment Variable Documentation*. Accessed: 2025-01-20. 2025. URL: https://pkg.go.dev/runtime#GOMAXPROCS.

[18] Go Standard Library. *Go encoding/csv Package*. Accessed: 2025-01-20. 2025. URL: https://pkg.go.dev/encoding/csv.

[19] Yaya Heryadi, Lili Ayu Wulandhari, Bahtiar Saleh Abbas, et al. "Recognizing debit card fraud transaction using CHAID and K-nearest neighbor: Indonesian Bank case". In: *2016 11th International Conference on Knowledge, Information and Creativity Support Systems (KICSS)*. IEEE. 2016, pp. 1–5.

[20] Obinna Iheanachor. *Wisabi Bank Dataset on Kaggle*. Accessed: 2025-01-20. 2025. URL: https://www.kaggle.com/datasets/obinnaiheanachor/wisabi-bank-dataset.

[21] Joho. *godotenv - GoDotEnv Package*. Accessed: 2025-01-20. 2025. URL: https://pkg.go.dev/github.com/joho/godotenv.

[22] Ramez Kian and Hadeel S Obaid. "Detection of fraud in banking transactions using big data clustering technique customer behavior indicators". In: *Journal of applied research on industrial engineering* 9.3 (2022), pp. 264–273.

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

[24] Daniel Lugosi Enes. "Concurrent Implementation of Multidimensional Range Queries". UPC, Facultat d'Informàtica de Barcelona, Departament de Ciències de la Computació, July 2019. URL: http://hdl.handle.net/2117/169246.

[25] Fernando Magdalena-Laorden. *Artificial intelligence and ontology applied to credit card fraud detection (Bachelor Thesis)*. Universitat Politécnica de Madrid - E.T.S. de Ingenieros Informáticos, 2021. URL: https://oa.upm.es/69050/.

[26] Fernando Martín-Canfrán et al. "Continuous Query Engine to Detect Anomalous ATM Transactions". In: *Proceedings of the Alberto Mendelzon Workshop, AMW 2024, Mexico City, México. To be published in CEUR*. 2024.

[27] Sanaz Nami and Mehdi Shajari. "Cost-sensitive payment card fraud detection based on dynamic random forest and k-nearest neighbors". In: *Expert Systems with Applications* 110 (June 2018). DOI: 10.1016/j.eswa.2018.06.011.

[28] Neo4j. *Bolt Application Protocol*. Accessed: 2025-01-20. 2025. URL: https://neo4j.com/docs/bolt/current/bolt/.

[29] Neo4j. *Cypher LOAD CSV Command*. Accessed: 2025-01-20. 2025. URL: https://neo4j.com/docs/cypher-manual/5/clauses/load-csv/.

[30] Neo4j. *Cypher Query Language*. Accessed: 2025-01-20. 2025. URL: https://neo4j.com/docs/cypher-manual/current/introduction/.

[31] Neo4j. *Neo4j Go Driver*. Accessed: 2025-01-11. 2025. URL: https://pkg.go.dev/github.com/neo4j/neo4j-go-driver/v5 (visited on 01/11/2025).

[32] Neo4j. *Neo4j Go Driver Manual*. Accessed: 2025-01-11. 2025. URL: https://neo4j.com/docs/go-manual/current/ (visited on 01/11/2025).

[33] Neo4j. *Neo4j Go Driver v5.24.0*. Accessed: 2025-01-20. 2025. URL: https://pkg.go.dev/github.com/neo4j/neo4j-go-driver/v5@v5.24.0/neo4j.

[34] CBT Nuggets. *Why is Go so good at concurrency?* Accessed: 2025-01-10. 2025. URL: https://www.cbtnuggets.com/blog/technology/programming/why-is-go-so-good-at-concurrency.

[35] Stack Overflow. *When should you use a mutex over a channel?* Accessed: 2025-01-10. 2025. URL: https://stackoverflow.com/questions/47312029/when-should-you-use-a-mutex-over-a-channel.

[36] Edelmira Pasarella et al. "A computational framework based on the dynamic pipeline approach". In: *Journal of Logical and Algebraic Methods in Programming* 139 (2024), p. 100966.

[37] Md Mijanur Rahman, Anuva Rani Saha, et al. "A comparative study and performance analysis of ATM card fraud detection techniques". In: *Journal of Information Security* 10.03 (2019), p. 188.

[38] RDLab, Universitat Politècnica de Catalunya. *RDLab-UPC Cluster*. Accessed: 2025-01-20. 2025. URL: https://rdlab.cs.upc.edu/.

[39] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013. ISBN: 1449356265.

[40] Jean Paul Rodriguez. *Apache Arrow*. 2023. URL: https://jeanpaulrodriguez.medium.com/apache-arrow-98f3c38ec875 (visited on 01/14/2025).

[41] Christopher Rost et al. "Seraph: Continuous Queries on Property Graph Streams". In: *EDBT*. 2024, pp. 234–247.

[42] Juan Pablo Royo-Sales. *An algorithm for incrementally enumerating bitriangles in large bipartite networks (Master Thesis)*. Universitat Politècnica de Catalunya - Barcelona Tech, 2021. URL: http://hdl.handle.net/2117/361615.

[43] Relia Software. *Concurrency in Golang*. Accessed: 2025-01-10. 2025. URL: https://reliasoftware.com/blog/concurrency-in-golang.

[44] SDM-TIB Team. *diefpy: A Python Library for Measuring the Efficiency of Continuous Query Execution*. Accessed: 2025-01-13. 2025. URL: https://sdm-tib.github.io/diefpy/.

[45] Diana Topol. *Apache Arrow and Go: A Match Made in Data*. 2022. URL: https://www.apachecon.com/acna2022/slides/01_Topol_Arrow_and_Go.pdf (visited on 01/14/2025).

[46] Umahmood. *Go Haversine Package*. Accessed: 2025-01-20. 2025. URL: https://github.com/umahmood/haversine.

[47] Unit21. *Card Cloning*. Accessed: 2025-01-04. n.d. URL: https://www.unit21.ai/fraud-aml-dictionary/card-cloning.

[48] Unknown. *Go and Apache Arrow: Building Blocks for Data Science*. 2018. URL: https://blog.gopheracademy.com/advent-2018/go-arrow/ (visited on 01/14/2025).

[49] Unknown. *Make Data Files Easier to Work With Using Golang and Apache Arrow*. 2023. URL: https://voltrondata.com/blog/make-data-files-easier-to-work-with-golang-arrow (visited on 01/14/2025).

[50] Anurag Verma. *Chunk-by-Chunk: Tackling Big Data with Efficient File Reading in Chunks*. 2023. URL: https://medium.com/@anuragv.1020/chunk-by-chunk-tackling-big-data-with-efficient-file-reading-in-chunks-c6f7cf153ccd (visited on 01/14/2025).

[51] Yelleti Vivek et al. *ATM Fraud Detection using Streaming Data Analytics*. 2023. arXiv: 2303.04946 [cs.LG]. URL: https://arxiv.org/abs/2303.04946.

[52] Go Wiki. *Go Wiki - use a mutex or channel?* Accessed: 2025-01-10. 2025. URL: https://go.dev/wiki/MutexOrChannel.

[53] Brian Wofford. *The Comprehensive Guide to Concurrency in Golang*. Accessed: 2025-01-10. 2025. URL: https://bwoff.medium.com/the-comprehensive-guide-to-concurrency-in-golang-aaa99f8bccf6.

[54] Carlo Zaniolo. "Logical foundations of continuous query languages for data streams". In: *Proceedings of the Second international conference on Datalog in Academia and Industry*. 2012, pp. 177–189.

[55] Carlo Zaniolo. "Logical foundations of continuous query languages for data streams". In: *International Datalog 2.0 Workshop*. Springer. 2012, pp. 177–189.