TFM-FernandoMartín

Fernando Martín Canfrán

January 14, 2025

1 Experimental Evaluation

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

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

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

To evaluate the $\mathsf{DP}_{\mathsf{ATM}}$ as a real-time system, we conducted two kinds of experiments. On the one hand the analysis of the $\mathsf{DP}_{\mathsf{ATM}}$ on a high-load stress scenario (E1), where we study the behavior of the system in a worst case scenario, receiving a high loaded transaction stream. On the other hand, the evaluation of the $\mathsf{DP}_{\mathsf{ATM}}$ in scenarios reflecting more real-world possible conditions, with a more realistic transaction frequency (E2). In each of the cases we intend to show the performance of the system under different possible configurations in terms of the number of filters and available cores on the running machine.

To evaluate the behavior of the system in these scenarios, we decided to analyze not only classical real-time system metrics but also newly proposed metrics for assessing the system's continuous behavior in terms of the emitted results. The selection of the metrics for the system evaluation is described in **??**. In **??** we explain the possible considered definitions for a system result, which we will select depending on the experiment.

The system will be tested for different possible bank sizes, with varying numbers of cards, ATMs, and stream sizes, simulating various time intervals of card-ATM interactions. The selection of the different possible bank and synthetic transaction stream sizes are given in **??**.

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

Finally, we devote **??** to 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 ??.

1.1 Definition of a System Result

In principle, attending only to our system description, we consider a result - or equivalently, an answer of the $\mathsf{DP}_{\mathsf{ATM}}$ - to be synonymous with an alert, caused by a positive fraud check on a *Filter* stage.

For experimental purposes, we propose an alternative result definition, in which we consider a result to be equal to a fraud pattern check. That is, all the fraud pattern checks 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 checks as results is done only due to experimental purposes, 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 checks as results will derive in a communication overhead between the *Filter* F stages and the *Sink* Sk stage, where the results are gathered. However, on a production version of the $\mathsf{DP}_{\mathsf{ATM}}$, we will utilize the original definition of a $\mathsf{DP}_{\mathsf{ATM}}$ system result - the alerts are the results - as there would be no interest in sending negative fraud pattern checks to Sk . Only the positive fraud pattern checks - the alerts - are of interest in that case, therefore reducing at maximum the overhead in the communication channel between the F's and Sk and the corresponding processing of results in Sk .

1.2 Evaluation Metrics

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

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

- **Response Time (`RT`) and Mean Response Time (`MRT`)**
  `RT` captures the time it takes for the system to emit a result. It is the elapsed

time from the moment an interaction arrives to the system until the result of its respective fraud pattern check is produced. A fine-grained description on how this metric was captured in the $\mathsf{DP}_{\mathsf{ATM}}$ system is included in **??**. The `MRT` is the average response time metric for all the results emitted by the system.

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

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

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

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

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

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

To obtain the `dief@t` and `dief@k` metrics we used the `diefpy` tool [**exps-diefpy-tool**], which calculates them from the generated answer/result traces by our system. This tool provides us with other utilities for the visualization of the metrics on the obtained sets of results such as: the visualization of answer traces, the generation of radar plots to compare the `dief@t` with the other conventional/classical metrics, the generation of radar plots to compare the `dief@k` at different answer completeness percentages, among others.

We additionally extended/modified the tool for our specific needs. This was done in order to visualize other metrics, especially the `mrt`, but also others like the thoughtput `T`, `interactions/s` or the `TFFT`.

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

1.3 Bank and Stream Configurations

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

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

Fernando: TODO: Poner mejor

Based on these references we constructed different streams with sizes and fraud ratio similar to the ones mentioned, using our generator of synthetic transactions, generated from different bank databases. The variation on the size of the simulated stream of transaction on which to test our system is not the unique we do, we also variate the bank database size in terms of the number of cards and ATMs it contains. This variation, apart from having an influence on the volume of generated transactions on a time interval when using our synthetic transaction stream generator, it is expected to have an influence on the performance of the system due to the expected added overhead when querying the Neo4j stable graph database. In our experiments we propose two bank database sizes; one small bank database with 2000 cards and 50 ATMs, and one large bank database with 500000 cards and 1000 ATMs.

Fernando: Poner referencias a bancos reales de tamaños similares?

When generating the streams, in order to obtain the desired stream sizes, we needed to consider that our transaction generator takes as base the behavior of the clients of the Wisabi Bank Database, where each client typically produces at most ~ 1

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

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

Fernando: Pongo esto?:Note that, for simplicity, we are assuming the number of bank branches as the number of ATMs and the number of clients as the number of cards.

E1: Evaluation in a High-Load Stress Scenario

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

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

Continuous delivery of results in a high loaded scenario

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

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

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

- **No real case simulation**
- **Measure the load the system can take**: for the different system variations given a same stream.
- **Diefficiency metrics**: since time arrival of the transactions to the system is now ignored, and all the transactions come one after the other, a result to be produced do not need to wait for the real timestamp of the transaction.

Therefore, we could see the differences in continuously delivering results of the different systems under the same input stream load (more clear than before).

IMPORTANT: WHAT DO WE WANT TO TEST?

Definition of the objectives of the experiments:

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

Problems derived:

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

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

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

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

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

1.3.2 Small bank size & small transaction stream

1-core

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

Results for executions with 1-core:

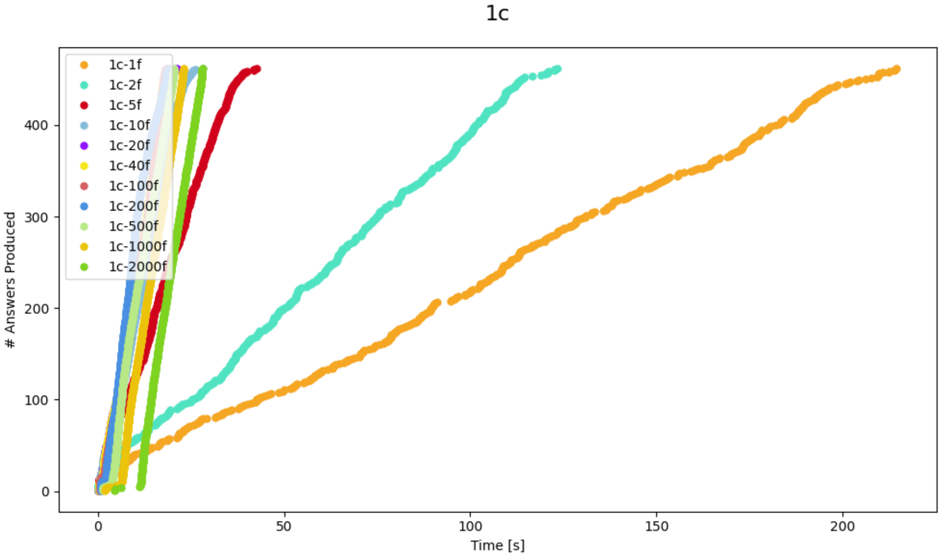


Figure 1: Alerts trace in time (s) - 1 core

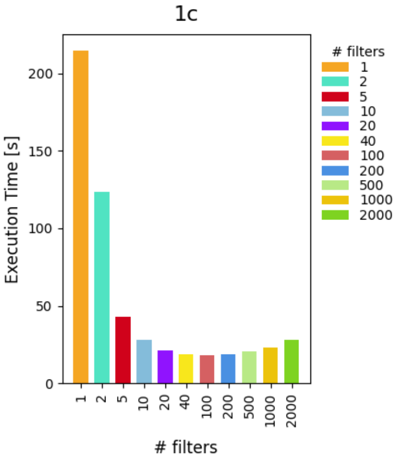


Figure 2: Execution time (s) - 1 core

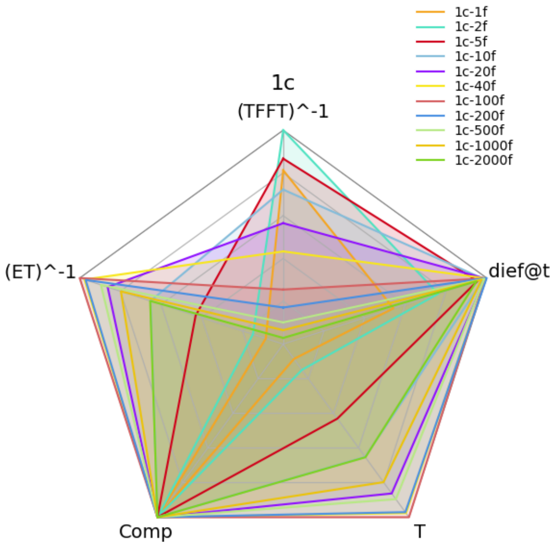


Figure 3: `dieft` radar - 1 core

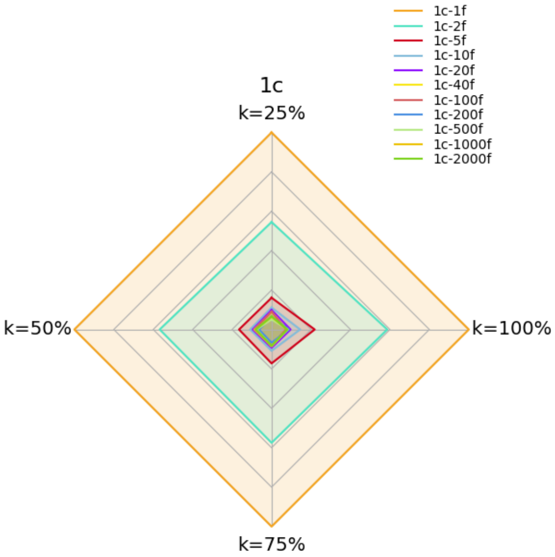


Figure 4: `diefk` radar - 1 core

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

4-cores

Results for executions with 4-core:

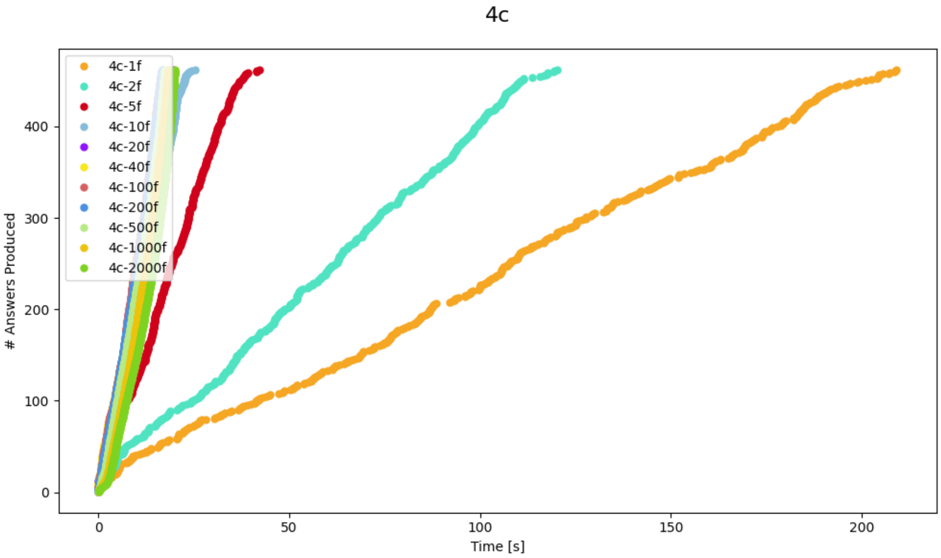


Figure 5: Alerts trace in time (s) - 4 core

8-cores

Results for executions with 8-core:

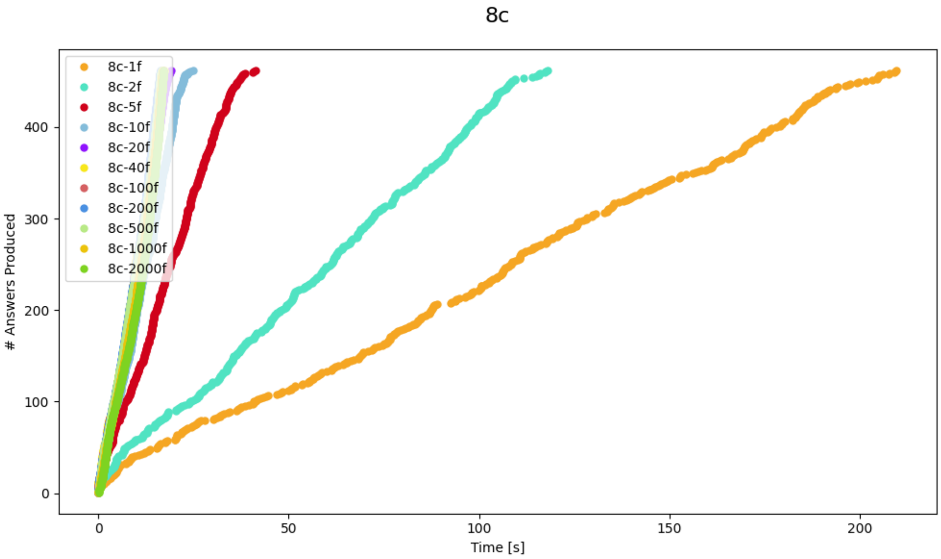


Figure 6: Alerts trace in time (s) - 8 core

16-cores

Results for executions with 16-core:

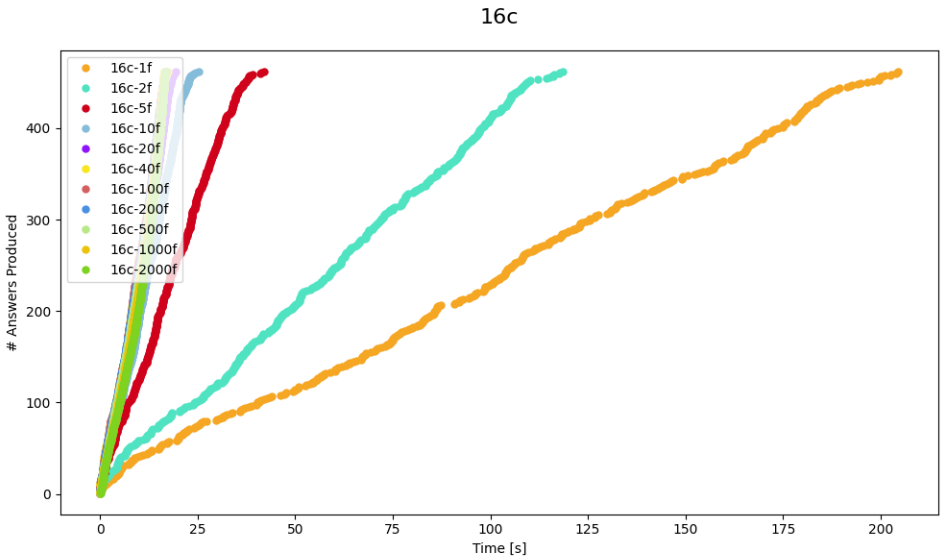
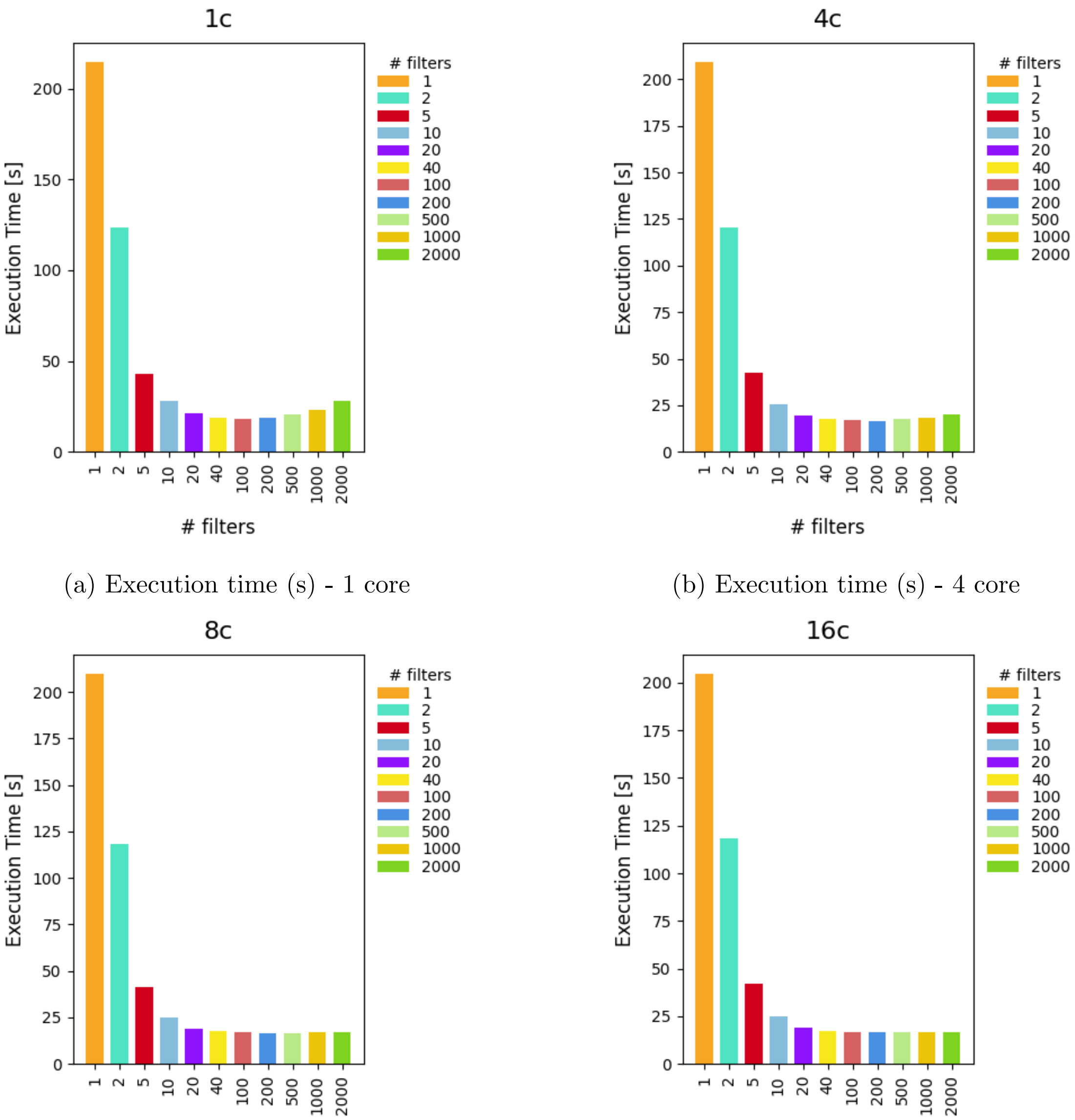


Figure 7: Alerts trace in time (s) - 16 core

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



(a) Execution time (s) - 1 core

(b) Execution time (s) - 4 core

(c) Execution time (s) - 8 core

(d) Execution time (s) - 16 core

Figure 8: Execution times for different core configurations.

Some observations:

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

Transaction stream - medium

- `NUM_DAYS` = 60

- anomalous_ratio = 0.02 (2%)

This setup gives us a transaction stream of

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

Transaction stream - big

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

This setup gives us a transaction stream of

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

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

→ Issue: Connection timeout
In the cases:

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

Possible fixes:

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

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

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

Some details about our Neo4j VM:

- 4 cores and 20GB of RAM
- No limit on the number of parallel sessions. However it seems that by default there is a limit on the number of parallel transactions to 1000.

E2: Evaluation in a Real-World Stress Scenario

E1: Mean Response Time

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

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

1.3.3 E1: Mean Response Time

Real time-event stream simulation

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

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

The consequences for the experiments and metrics:

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

Problems derived to pay attention to:

- Shrinking the timestamps to a smaller time interval, produces the emergence of not real fraud patterns that before did not exist due to their real and "correct" larger time distance. Example:
  - Consider the original size of the time interval of the input stream $T = 120h$ (5 days) and $T' = 24h$.
  - Consider two consecutive regular transactions of a certain client performed in two different ATMs `ATM-x` and `ATM-y` with `t_min`= 8h (minimum time difference to traverse the distance from `ATM-x` to `ATM-y`) and `t_diff`= 24h (time difference between the first and the second transaction).

  - → Note that with the scaling the time difference `t_diff` would be of 5 times less, that is, `t_diff` $= 4.8h$. Therefore this will make `t_diff'` $= 4.8h <$ `t_min` $= 8h$.

- → (*) Solution A: **introduce the scaling factor as a input parameter** and consider it also for the fraud checking so to properly **scale the `t_min` variable** (`t_min` $= 8h \rightarrow$ `t_min'` $= \frac{8}{5}h = 1.6h$) and therefore:
  - Before scaling: `t_diff` $= 24h >$ `t_min` $= 8h$.
  - After scaling (scale factor $= \frac{1}{5}$): `t_diff` $= 24 * \frac{1}{5} = 4.8h >$ `t_min` $= 8 * \frac{1}{5} = 1.6h$.

- → Solution B: conserve the original timestamps, and consider the mapped-reduced timestamps for simulating the arrival times of the transactions into the system while taking the original timestamps for the checking of the frauds.

IMPORTANT: WHAT DO WE WANT TO TEST?

Definition of the objectives of the experiments:

- See and compare the behavior of the system(s) with different streams (different number of cards, greater or smaller size of the bank - and therefore its database).

  - Alert/result response time comparison. **Continuous delivery of results (diefficiency metrics) does not make sense!**. With the objective to see that we can see lower response time in the case of the dp versions.

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

Problems derived:

- **Continuous delivery of results comparison does not make sense.** → In a real time simulation, for any system, results can only be emitted whenever the corresponding anomalous transaction a_i reaches the system. That happens at the same time t_i for both approaches when the input stream is simulated at real time, meaning that the result corresponding to the anomalous transaction a_i can not be emitted in any case before time t_i. Therefore, the difference in time delivery of this result between the different approaches is not expected to be high unless we make the systems to be loaded enough. **Therefore, for small sized banks this does not really make sense...**

- **Losing of alerts**: Due to scaling we are losing alerts since we have seconds precision. We will have to scale to the millisecond or nanosecond the timestamps to possibly do not loose those alerts, due to time scaling precision.

- **Although scaling, the load we are simulating is higher than real... - like for the not real time approach**

1.3.4 Initial experiments

Small initial graph database (gdb) size:

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

Transaction stream:

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

This setup gives us a transaction stream of

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

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

Table 1: Different experimental setups results

Some nomenclature:

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

Some results:

1h scaling

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

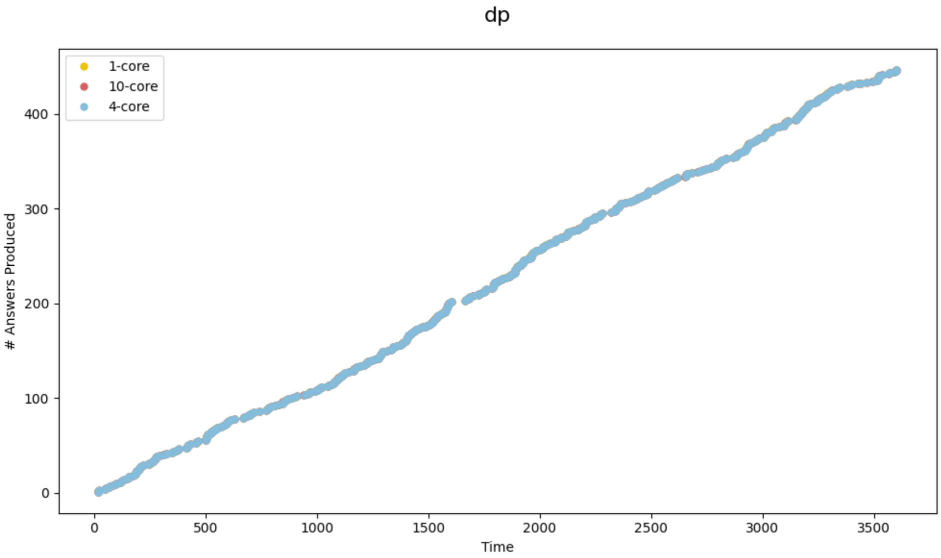


Figure 9: Trace 1h

Only for the first 10 results (alerts):

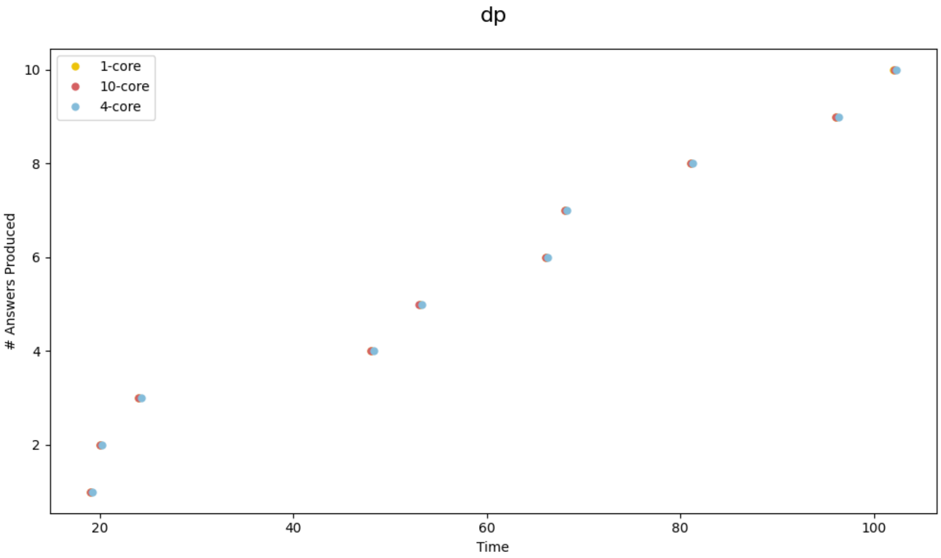


Figure 10: Trace 1h - first 10 alerts

6h scaling

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

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

1.4 Definitive Experiments

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

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

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

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

1.4.1 Bank size: Initial - Small

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

Transaction stream - small

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

This setup gives us a transaction stream of

- `total_tx` = 39959

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

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

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

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

1.5 E1: Mean Response Time

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

Setting up the experiments:

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

1.5.1 Small bank size & small transaction stream

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

My doubts on these experiments

- Results = Checks: same issue as with the alerts, a same check(tx1,tx2) on a card can not be performed until the tx2 reaches the system, which is the same simulated time on any system.
- For a small bank size, as the simulation tends to be closer to reality (lower scaling), then the system has lower overhead and the differences between the different variations will be almost negligible. Also the time needed to perform these simulations increases.
- However, on the contrary, when the scaling is high, then the simulation is farther from reality, making the system to be more loaded than in a real case scenario, reaching a point that the experiment will be almost the same as E2 (reading the transaction stream input directly without any real-time simulation).
- To have a "real-time" simulation in which differences between the systems can be observed we will need to simulate a big bank, which is able to provide a dense/big stream, which intuitively, even more if the stream is scaled, the simulation will be almost like reading the stream directly from the input like without any "real-time" simulation, like in the case of the E2 experiments.

Proposals:

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

Some experiments that were tested:

- Scaled small bank data stream of 30 days to 10 minutes.

- Tested for these combinations:

# cores	# filters
1	1
4	40
16	2000

1.6 Update: Recording all the checks

1.7 How to measure MRT

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

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

So far:

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

Some experiments were done to inspect whether there was a significant difference on the measurement position. In particular, we measured the response time of each of the checks both in the Filter and in the Sink, to finally obtain the mean response metrics times of both approaches. In the table **??** we show the mean response time metric in ms measured at the Sink (`mrt-Sink`) and the difference of this measurement on average with respect to the measurement done in the Filter in ms (`mrt-difference`).
Note that each of this experiments were done running in not-real-time, with the small bank size, and the `30-0.02` stream. Each of the experiments were run 10 times... 16GB RAM...

As it can be observe the differences are negligible, and therefore for simplicity, and to maintain the "philosophy" of the dynamic pipeline, we decided to keep the measurement of the response times of the checks on the Sink stage.
Note that, measuring in the Filter would imply assuming that the alerts had to be sent from there, to be realistic to where we are measuring. Note that: measuring all the results and not only the alerts imply unnecesarily overloading the system, since only sending the alerts to the Sink should be enough for the purposes of our application. However, due to experimental purposes we are forced to send/measure all the checks, in order to be able to compare the continuous delivery of results of all the system configurations/variations.

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

Table 2: Comparison of the response time measurement positions with different system configurations

1.7.1 Bank size: Medium

For these experiments, to generate the stream of tx, we needed to simplify this process in order to be able to generate a stream in a feasible amount of time. In particular we used the simplifed version of the `txGenerator.py`: `txGenerator-simplified.py` → with a random ATM-subset instead of a closest to client ATM-subset. Also variation on the transaction distribution times.

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

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

Transaction stream - small

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

This setup gives us a transaction stream of

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

First run:

- 16GB RAM
- 16 cores
- Experiments for $|filters| \geq 100$
- x1 run each job

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

TO TRY:

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

1.8 How to run the experiments

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

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

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

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

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

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

1.9 Input reading by chunks

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, we implemented two different cases of simulations. The real-case scenario and the high loaded test scenario. In the first case, the interactions, although read by a file of artificial simulated interactions, are provided to the pipeline data stream in such a way that they simulate their actual arrival time to the system, with the corresponding time separation between them. In the second case, the interactions are provided just one after the other as fast as possible as they are read.

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

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

Listing 2: `csv-bufio` reader

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

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

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

1.9.1 Apache Arrow

Apache arrow CSV package allows reading csv in chunks of n rows, called *records*. The thing is that *records* / apache arrow is optimized storing the data in a columnar way (by columns). So that we can not access the original n rows easily, but instead the columns of these rows. And therefore, from them we will need to reconstruct

the rows by taking the corresponding elements from each of the columns, given the index of the corresponding row.
Good references:

- Apache Arrow and Go - Good tutorial

1.9.2 encoding/csv

1.9.3 Experiments over the different approaches

Approaches:

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

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

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

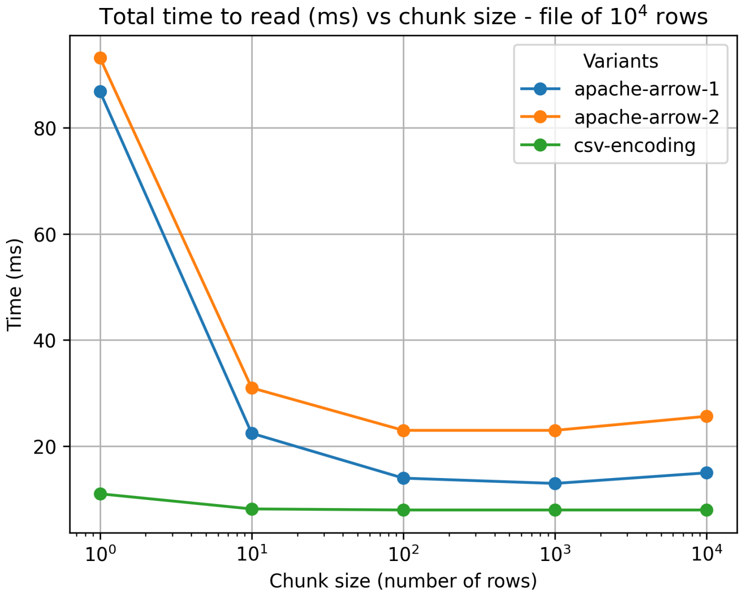


Figure 11: Comparison of the variants for file of 10^4 rows

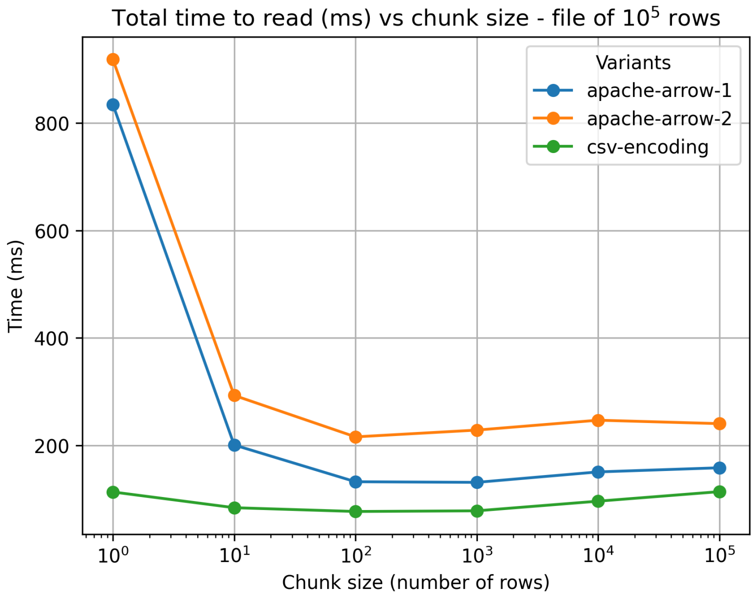


Figure 12: Comparison of the variants for file of 10^5 rows

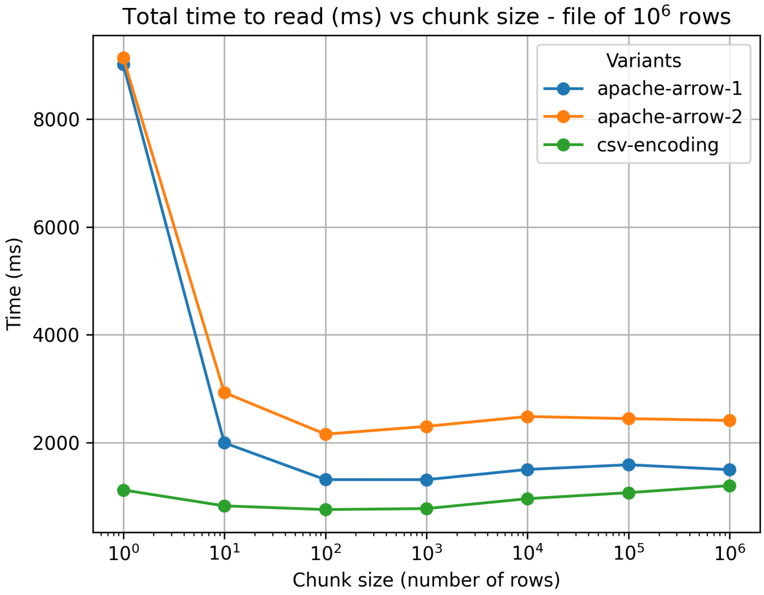


Figure 13: Comparison of the variants for file of 10^6 rows

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

- Compare the variant with worker and chunk size of 10^2 with the one without worker and therefore not reading by chunks.

- Comparison for different sizes of files: 10^4, 10^5 and 10^6 number of rows (transactions).
- Each of the experiments is done 20 times to obtain stable measurements.

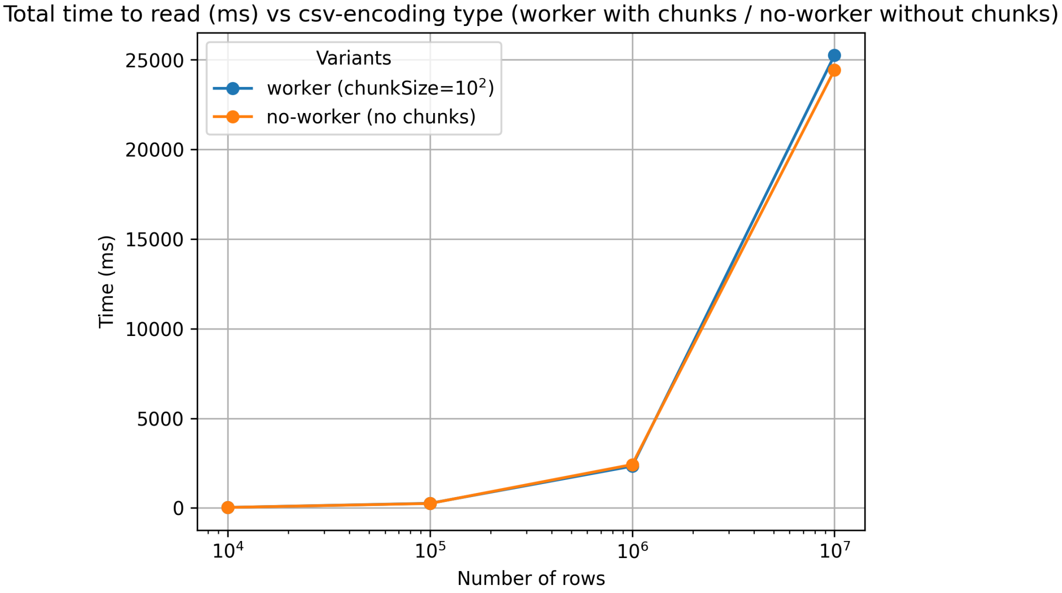


Figure 14: Comparison of `csv/encoding` variants up to 10^7 rows

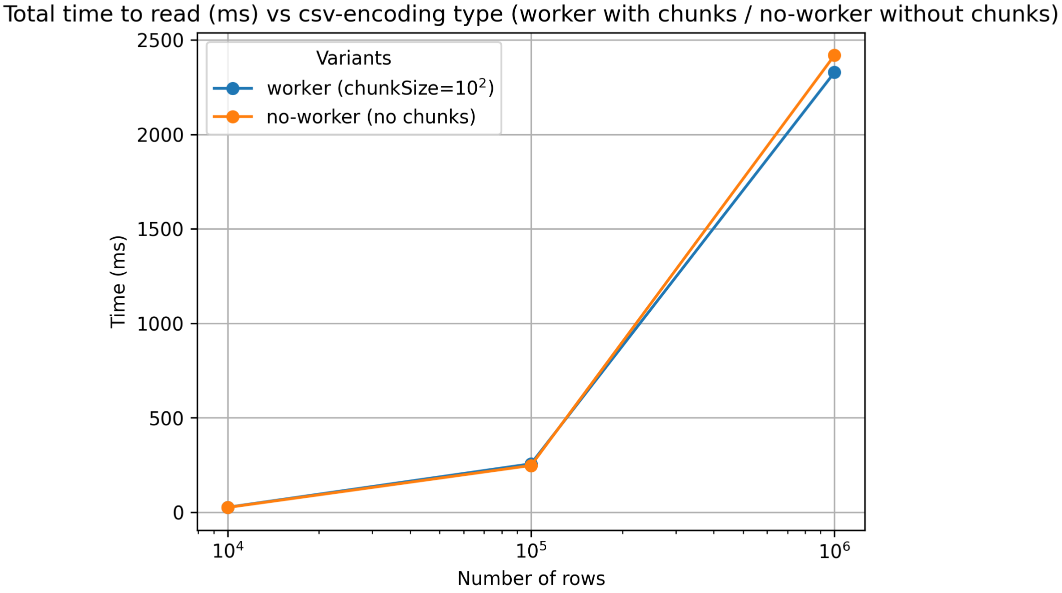


Figure 15: Comparison of `csv/encoding` variants up to 10^6 rows

- Differences insignificant
- Depend on the application

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

As it can be seen, the differences are insignificant, and the selection of each of the variants will depend on the application. For example, we suspect that the worker version can be beneficial in the real time simulation, so that we do not make the reading be the bottleneck of the simulation, by having a background process reading input transactions from the stream files while the main process providing the input to the pipeline can be stopped doing the real time simulation.
TODO: Comparativa run same experiment NRT with worker VS without worker for the input providing - a single experiment example to show that is better with worker!
Some (other) references:

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