1 Experiments

Note that, since 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).

1.1 1st option: Real time-event stream simulation

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

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

The consequences for the experiments and metrics:

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

Problems derived to pay attention to:

- Shrinking the timestamps to a smaller time interval, produces the emergence of not real fraud patterns that before did not exist due to their real and "correct" larger time distance. Example:

  - Consider the original size of the time interval of the input stream $T = 120h$ (5 days) and $T' = 24h$.
  - Consider two consecutive regular transactions of a certain client performed in two different ATMs `ATM-x` and `ATM-y` with `t_min`= 8h (minimum time difference to traverse the distance from `ATM-x` to `ATM-y`) and `t_diff`= 24h (time difference between the first and the second transaction).
  - → Note that with the scaling the time difference `t_diff` would be of 5 times less, that is, `t_diff` $= 4.8h$. Therefore this will make `t_diff'` $= 4.8h <$ `t_min` $= 8h$.

- → (*) Solution A: **introduce the scaling factor as a input parameter** and consider it also for the fraud checking so to properly **scale the `t_min` variable** (`t_min` $= 8h \rightarrow$ `t_min'` $= \frac{8}{5}h = 1.6h$) and therefore:
  - Before scaling: `t_diff` $= 24h >$ `t_min` $= 8h$.
  - After scaling (scale factor $= \frac{1}{5}$): `t_diff` $= 24 * \frac{1}{5} = 4.8h >$ `t_min` $= 8 * \frac{1}{5} = 1.6h$.
- → Solution B: conserve the original timestamps, and consider the mapped-reduced timestamps for simulating the arrival times of the transactions into the system while taking the original timestamps for the checking of the frauds.

1.2 2nd option: real timestamp omission

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

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

Some (other) references:

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

2 Experiments description

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

- $|ATM| \sim 200$
- $|Card| \sim 14000$

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

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

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

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

2.1 Initial setup

Small initial graph database (gdb) size:

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

Transaction stream:

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

This setup gives us a transaction stream of

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

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

Table 1: Different experimental setups results

Some nomenclature:

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

Some results:

2.1.1 1h scaling

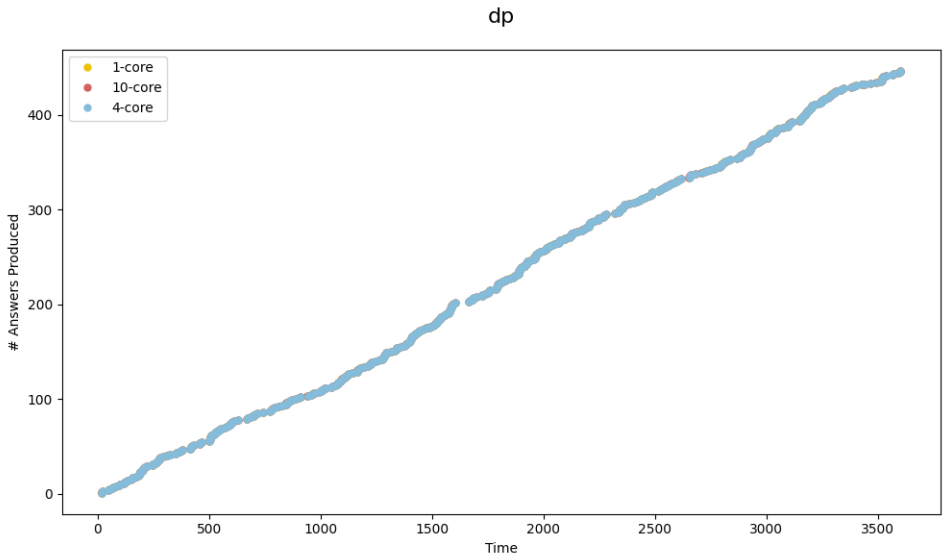

Figure 1: Trace 1h

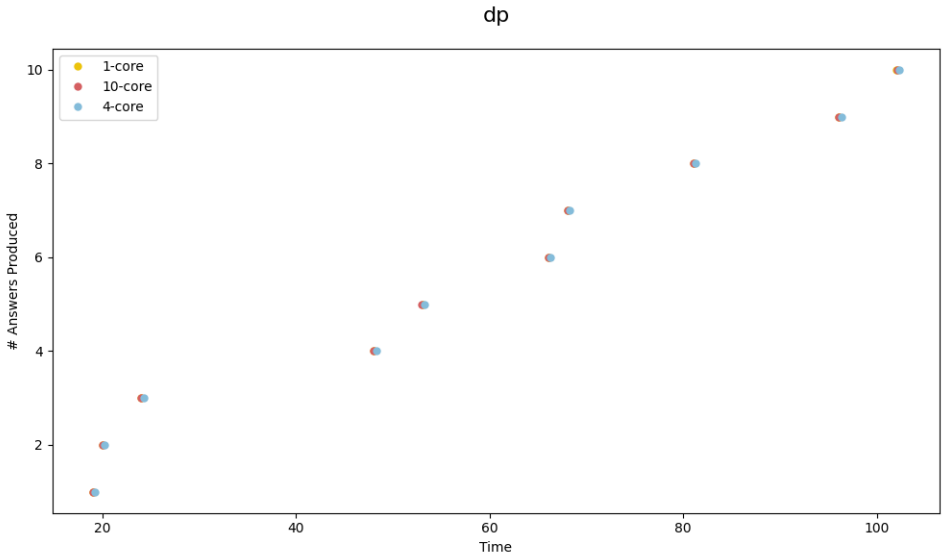

Figure 2: Trace 1h - first 10 alerts

2.1.2 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 omitted 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.

2.2 Bigger instances

3 Input reading by chunks

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

3.1 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

3.2 encoding/csv

3.3 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).

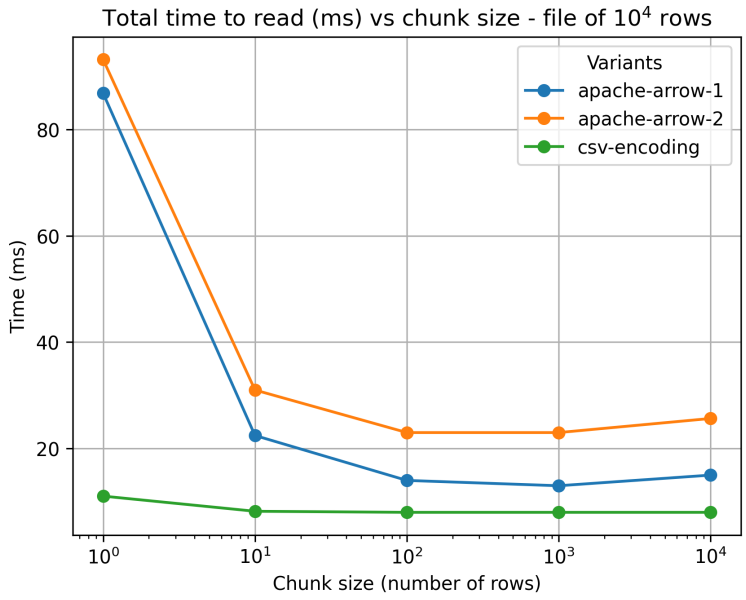

Figure 3: Comparison of the variants for file of 10^4 rows

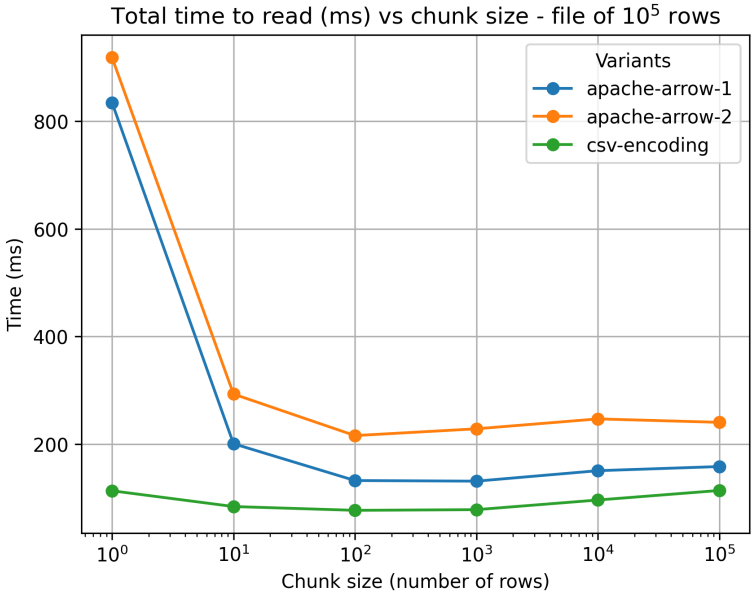

Figure 4: Comparison of the variants for file of 10^5 rows

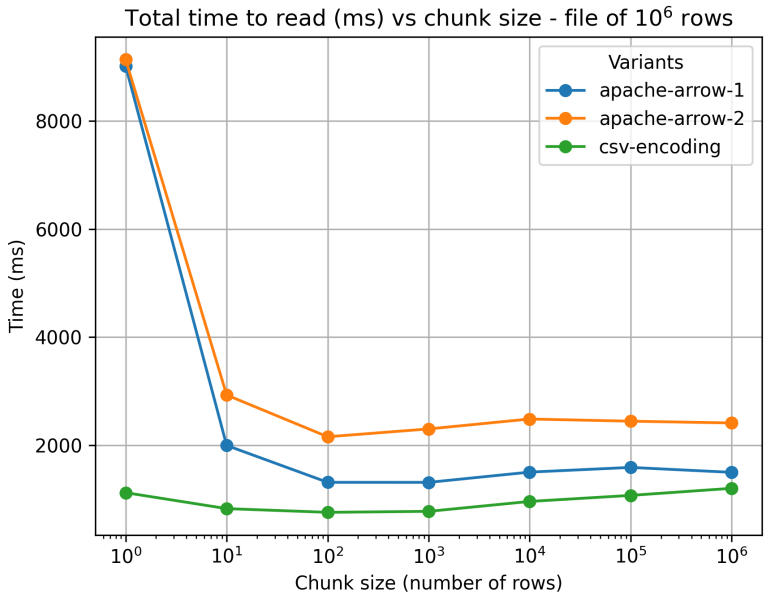

Figure 5: Comparison of the variants for file of 10^6 rows

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).

[1]

References

[1] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.