1 Dynamic Pipeline

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

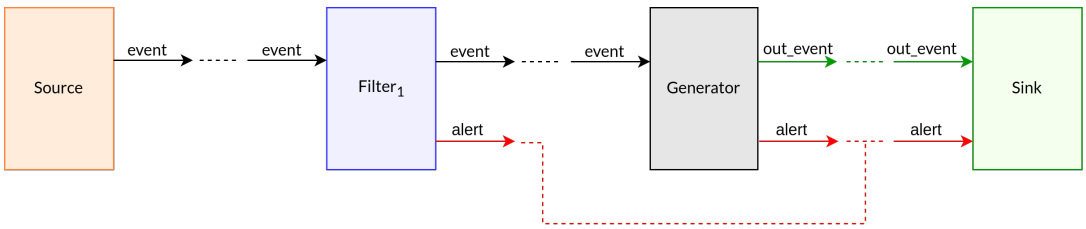

Figure 1: Pipeline Schema

Description of the channels:

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

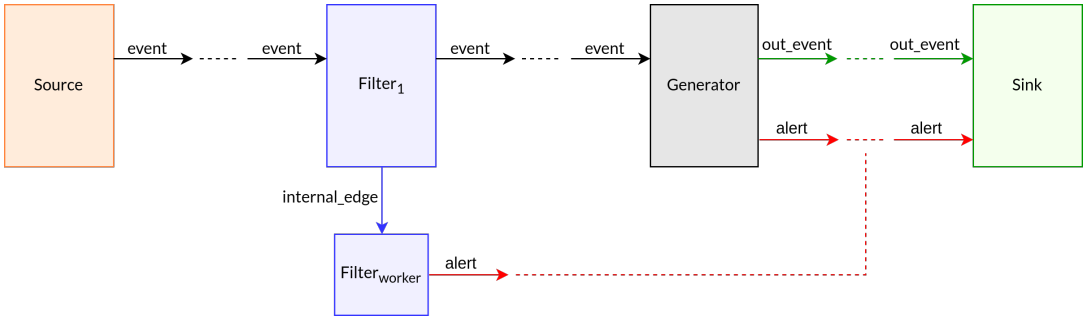

Figure 2: Pipeline Schema with Filter detail

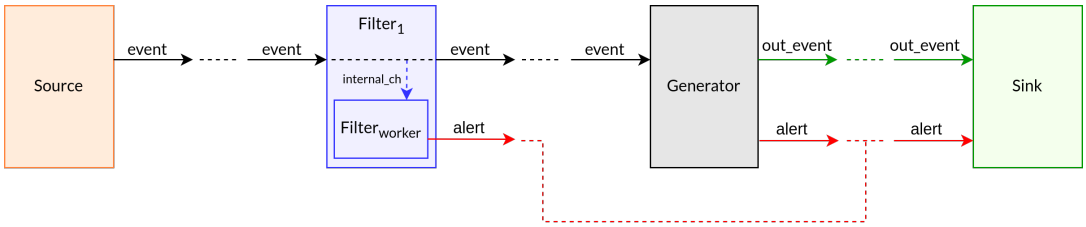

Figure 3: Pipeline Schema with Filter detail 1

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

Some notes on the implementation decisions:

1.1 Filter worker

Options:

- External (named) goroutine
- → Internal anonymous goroutine

Advantages of this decision:

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

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

- Shared buffer using mutex
- → Channel

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

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

Some links:

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

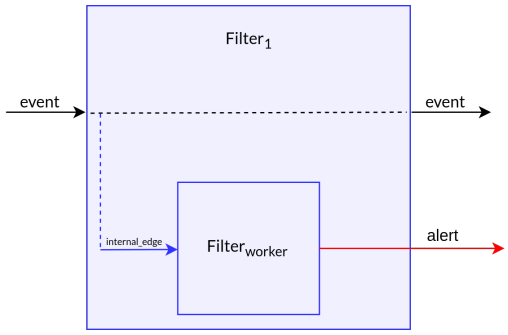

Figure 4: Filter Worker detail

2 Fraud Patterns

2.1 Fraud Pattern I - Card Cloning

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

Algorithm 1 CheckFraud(S, e_{new}) – **initial version**

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

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

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

- **Pairwise detection**. The checking of the anomalous fraud scenario is done doing the check between the new incoming edge e_{new} and each of the edges e_i of the filter's subgraph S.
- **Backwards order checking**. The pairs (e_{new}, e_i) are checked in a backwards traversal order of the edge list of the subgraph S, starting with the most recent edge of the subgraph and ending with the oldest.
- **Stop the checking whenever the first anomalous scenario is detected**. Whenever an anomalous scenario corresponding to a pair (e_{new}, e_i), then we stop the checking at this point and emit the corresponding alert. Therefore we do not continue the checking with previous edges of S.
- **Emission of the pair** (e_{new}, e_i) **as the alert**. The alert is composed by the pair (e_{new}, e_i) that is detected to cause the anomalous scenario. Both edges are emitted in the alert since we do not know which is the one that is the anomalous. On the one hand, it can be e_i, which is previous to e_{new}, in the case that e_i at the moment it arrived it did not cause any alert with the previous edges/transactions of the subgraph and it causes it now with a new incoming edge e_{new} which is a regular transaction of the client. On the other hand, it can be e_{new}, which is the last having arrived to the system, that it directly causes the alert with the last (ordinary) transaction of the card.

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

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

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

```
1: last ← |S|
2: e_last ← S[last]
3: t_min ← obtain_t_min(e_last, e_new)
4: t_diff ← e_new.start − e_last.end
5: if t_diff < t_min then
6:     createAlert(e_last, e_new)
7: end if
```

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

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

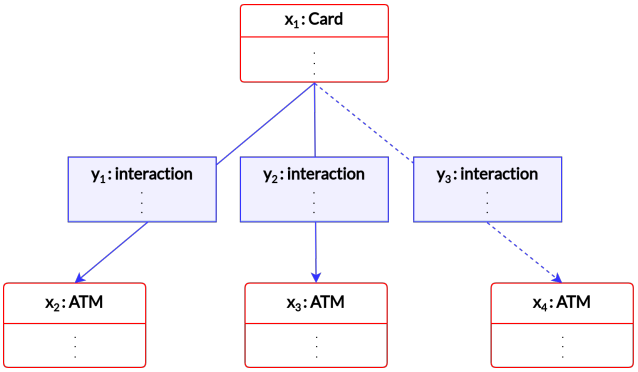

Figure 5: Subgraph S of a card – Fraud Pattern I

Note that we can have that:

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

- $\neg F_I(y_2, y_3)$: We analyze whether it would be interesting or not to continue the checking with previous edges of the subgraph, based on assumptions on the fraud checking between previous edges. In particular we can have two cases:
  - If $F_I(y_1, y_2)$: Having this it can happen that either $F_I(y_1, y_3)$ or $\neg F_I(y_1, y_3)$. In the case of $F_I(y_1, y_3)$, since $\neg F_I(y_2, y_3)$, we can infer that the anomalous scenario detected between y_1 and y_3 is a continuation of the same previous anomalous scenario detected between y_1 and y_2. Therefore, we can conclude that this does not constitute a new anomalous scenario that would require an alert.
  - If $\neg F_I(y_1, y_2)$: It can be shown that *by transitivity*, having $\neg F_I(y_1, y_2) \wedge \neg F_I(y_2, y_3) \implies \neg F_I(y_1, y_3)$.
    TODO: Show a formal demostration of this case!

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

TODO: Complete other aspects of the filter worker algorithmic workflow
Others – not so much related with the CheckFraud algorithm, but in general with the filter's algorithm –:

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

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

3 Connection to GDB

TODO: Add this in the data model section under a new subsection?
Some details / notes on how this is performed in golang.
So far:

- `DriverWithContext` object: only 1, shared among all the threads. It allows connections and creation of sessions. These objects are immutable, thread-safe, and fairly expensive to create, so your application should only create one instance.
- `Sessions`: so far we create one session every time we do a `checkFraud()` operation. Session creation is a lightweight operation, so sessions can be created and destroyed without significant cost. Always close sessions when you are done with them. They are not thread safe: you can share the main DriverWithContext object across threads, but make sure each routine creates its own sessions.

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

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

Some notes on this:

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

4 Experiments - Simulation with time-event stream

Note that, since the way we did the transaction generator, 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 size for the generated transaction stream would be having t around some weeks or month(s).

4.1 1st option: reduced time scaling

Take the stream of a certain time interval size t and map it into a smaller time interval t' where $t' << t$ so that:

- **Shorter experimental time**: Reduced time to test the system behavior. Instead of t, only t' time to test it.
- **Stress testing**: 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.
- **Graph database size VS amount of filters' subgraphs**: Although we simulate a higher load, the number of filters' subgraph corresponds to c the number of cards. 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 the differences.

- **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.

4.2 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.