MEM GRAPH

Fraud detection

WHITEPAPAPER

Graph Technology in Fraud Detection

Table of Contents

Introduction

Data is the most important asset an insurance company possesses, and it's important to keep it safe and handle it with care. Unfortunately, being cautious sometimes slows down innovation. In insurance, the go-to storage and analytics tech is still bundled into an old-fashioned relational database, considered state-of-the-art data management software.

This is exactly where graph technology shines in, social processes with complex interactions. Graph databases make the information system more responsive, allow for unexpected changes in the data structure and enable the insurance companies to successfully expand their business by tapping into the insights provided by the recommendations engines built on top of the data.

By using Memgraph for data storage and analytics, you can easily harness the power of graph databases to manage highly interconnected or networked data better, run faster queries, and use graph algorithms to gain better insights into the data.

Three Reasons Why Graph Databases Can Enhance the Insurance Industry

Good, old-fashioned relational databases are the industry standard and the go-to technology used for data storage in the insurance industry. All developers are familiar with using table data and SQL, so it is easy to think and safe to assume that tabular data is the most convenient way to organize data in the insurance industry. But what if we told you that is completely wrong?
Here are the three main reasons why shifting to graph databases should be a priority for any insurance company that wants to make their business more effective in detecting fraud and why it gives a helping hand with translating data trapped inside tables into graph objects.

1. Graph databases perform significantly faster with highly interconnected data

Relational databases use indexes, which are an expensive but indispensable tool that helps avoid searching for records row by row using a key. To identify a relationship between objects, two tables are joined and indexes on both tables are scanned

recursively to find all the elements fitting the query criteria. All these scans make JOIN operations in relational databases highly time-consuming. For example, a benchmark showing this comparison was made: given 1000 users with an average of 50 “friend” relationships, find if 2 people are connected in 4 or fewer hops. The query in a popular relational database took around 2000 ms.

On the other side, in graph technology, relationships between objects are one of its two main building blocks. They are created instantly when the data gets imported and connected to other objects. Instead of doing redundant recursive scans, relationships are found by doing graph traversals which are much faster. The same query that took the relational database 2000 ms, graph databases would complete in 0.4 ms.
When the example was scaled to a use case with 1,000,000 users, the relational database was stopped after several days of waiting for the results. The graph database? It took 2 ms to calculate the query output.

for a set of million users, the query was stopped after a few days

Indexes still exist in graph databases as well - they are not forbidden. They are supported and can be explicitly defined and making querying even faster.

The animation here shows the difference between connecting entities in relational and graph databases.

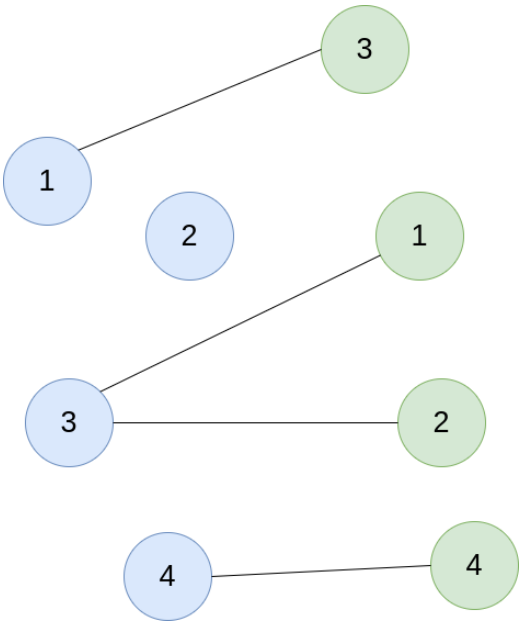

Figure 1. JOIN operation compared with graph traversals

To do this in a relational database, we combine the two tables by matching a foreign key to another table's key. On the other hand, in a graph database, we simply follow the relationships from one node to another.

By default, agents from insurance companies manually inspect fraud claims. Often, they can make little sense of the event by looking at the data in the table format due to many variables. This task consumes a lot of time and energy, even for skilled and trained professionals. So, not only is it not effective, but it also costs a lot of money eventually. If the data is modeled as a graph, the event would be instantly clear just by looking at the relationships between entities.

Because of the network-like representation of data in the graph databases, individual insurance claims are more easily understood when inspecting a graph rather than inspecting the endless rows of multiple tables.

Figure 3. shows how graphs can be visualized so we can easily understand the connections between the data. On the other hand, tabular data (Figure 2.) is much harder to understand at a glance.

CLAIM

clm_id	amount	fraud?	inc_id
1	500	False	2
2	7000	False	2
...			
112	1200	False	47

INCIDENT

inc_id	accident_date	file_date	pol_id	add_id
1	12.03.2002.	12.03.2002.	36	712
2	02.12.2004.	03.12.2004.	66	1123
...				
47	01.01.2021.	03.01.2021.	54	312

INJURY

inj_id	type	clm_id	ind_id
1	Body	101	32
2	Legs	3	12
...			
56	Legs	45	21

POLICY

pol_id	start_date	end_date	type	premium	insurer_id	insured_with_id	veh_id	add_id
1	12.04.2021.	12.04.2022.	Collision	D	3	5	32	123
2	30.10.2021.	30.10.2022.	Liability	null	4	8	3	133
...								
99	01.02.2010.	01.02.2015.	Liability	null	45	40	156	8

INCIDENT_INDIVIDUAL

inc_id	ind_id	relation
1	1	Driver
2	1	Injured
...		
200	78	Injured

ADRESS

add_id	street_name	city	country
1	Helsinki put	ZG	Croatia
2	Broadway	NY	USA
...			
221	No Lane	Nowhere	NL

VEHICLE

veh_id	type	model	year	price	owner_no
1	Vehicle	Toyota Yaris	2012	20000	1
2	Vehicle	Honda Civic	2022	35000	1
...					
123	Vehicle	BMW M3	2005	45000	4

INDIVIDUALS

ind_id	first_name	last_name	driver_licence	years_experience	phone	add_id
1	Lucius	Malfoy	TRUE	0	555-1234	10
2	Sovereign	Guvernor	TRUE	12	555-0154	6
...						
45	Steve	Ditko	TRUE	45	555-4575	220

CLAIM_PAYMENT

pay_id	amount	clm_id	payer_id	payee_id
1	5000	5	4	3
2	4000	21	51	24
...				
55	124000	22	44	21

Figure 2. relational database scheme

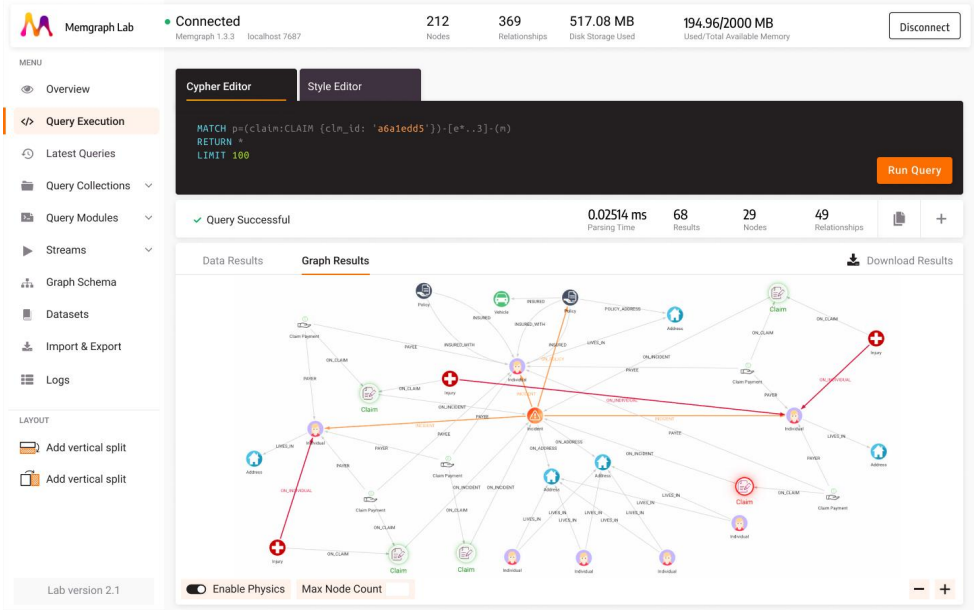

Figure 3. Visualization of graph data and how easy it is to see connections between data points

2. Graph data model can be easily changed

The world is filled with battered developers that survived painful schema changes. On the other side, graph developers are left unbruised and painless. The reason for that is that graph databases easily mimic unexpected changes, and their model schema can be changed easily.

In the example shown in Figure 4., let's say we have entities Individual and Vehicle. When building the relational table database system, Individual and Vehicle were modeled with separate tables containing their own IDs and other fields such as names. Individual also has a field veh_id (vehicle ID) which is a foreign key showing which vehicle the individual has insured. This is not the best practice for modeling this connection, but it is the one that could have been made in real life, and we used it in this example to show how complications might arise in relational tables.

Let's say that later, an individual wants to add another vehicle ownership to his name to be insured. Also, that vehicle has a lease on it and we want to model this information in our database as well.
Now, to store the lease, we need to model an entire database table, think through which information we want to store and could possibly store in the future, to make table columns.
Secondly, we have no way of recording multiple vehicle ids to a single individual. We should make another new table to model the connections between vehicles and individuals, requiring even more work.

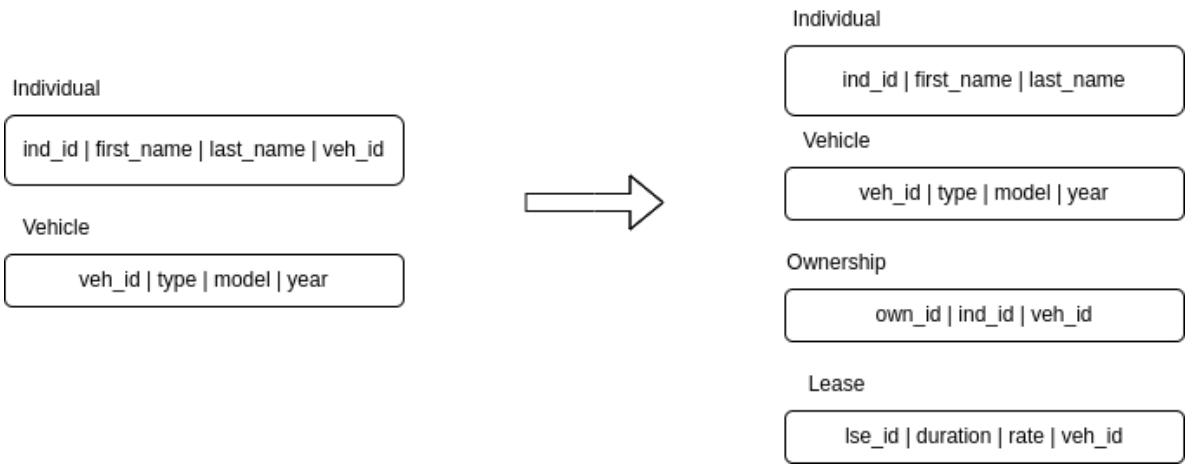

Figure 4. Changes to the relational database schema from adding information about a secondary vehicle and its lease

In a graph database (Figure 5.), there is no reason to fuss. Two new nodes are created, one with the label Vehicle and another with the label Lease (a label that previously didn't exist!). The Individual node is connected to the second Vehicle, as well as the Lease node to the Vehicle node. Done. Easy. All information about the lease and the vehicle is stored as properties in their respective nodes (and can sometimes be modeled as separate nodes).

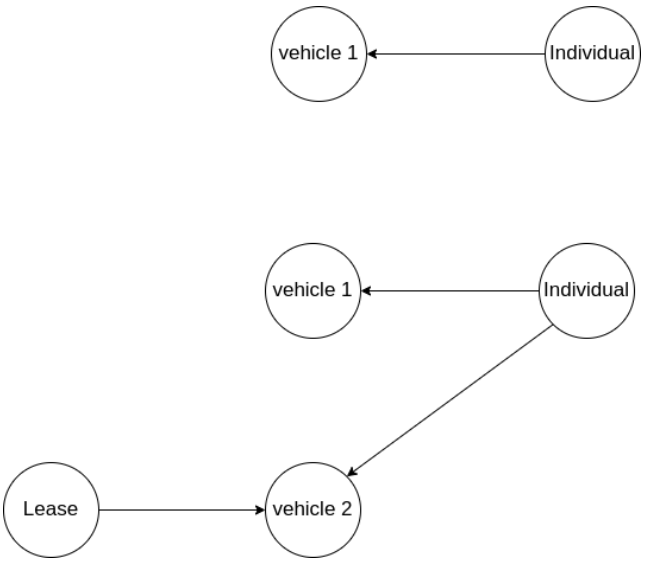

Figure 5. New nodes added to store additional data

3. Tight connections and patterns between entities can help design recommendation and fraud detection systems in insurance

Recommendation systems are not just used to suggest to online shoppers or Netflix viewers what else they might like. They are a powerful tool that can help companies gain future customers by tailoring insurance policies to their specific needs and recommending different policy options.

By analyzing the relationships, graph algorithms are applied to gain more features and information about nodes to work with. For example, a community detection algorithm assigns communities to nodes and pinpoints similarities in those communities, and PageRank measures the influence of a node. Using such information about the network (graph) around each node enhances recommendation systems, calculates similarities, and detects fraud better.

Figure 6. shows a machine learning system for detecting fraud which uses an enhanced feature set consisting of data gained with graph technology. You can read more about how to design such a system here (link to that reason).

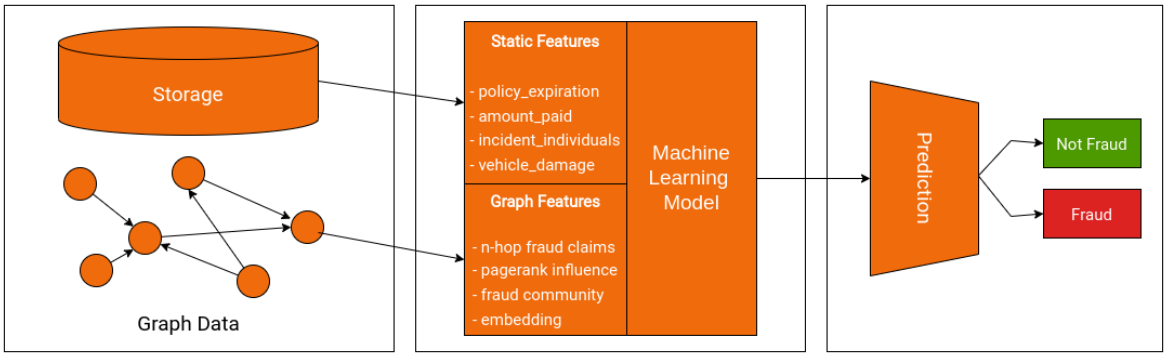

Figure 6. Schema of a machine learning system using graph features along with static features to detect fraud

How To Model Insurance Data As A Graph

Graph-based systems are faster and more flexible and are indispensable in recommendation and fraud detection systems that increase customer care and mitigate risks.
The first step to switching to a graph database is to update the data model. Although transforming data from a relational database into graph data seems insurmountable, it is not!
The following example from the insurance domain helps to switch the thinking about data organization as rows of records to graph objects like nodes and relationships. Use it as a springboard to move away from antiquated systems and get your company ahead of the competition.

Insurance data system with relational tables

To fully grasp the idea of table-to-graph transformation, one must understand the data domain.
We will be dealing with an example model of insurance company data. We model the insurance entities from a brief description:

“An individual with an insurance policy got involved in an incident with his vehicle along with a few other individuals. Some individuals sustained injuries. The insurance claim has been created in the system and a claim payment has been made to the insured individual”

Bolded words are modeled as tables, along with addresses, and specific information such as names, vehicle brands, incident dates, and more are table columns.

Data in this form is hard to analyze at first glance and extract any kind of useful knowledge from. Although the data is related across tables via foreign keys, the connections are not easily noticeable. If we want to query data between multiple tables to fetch all individuals involved in fraudulent claims, for example, we would need to use time-demanding JOIN operations. These data connections are made very easily in graph databases by traversing relationships.

Modeling comes first

Graph models really shine when it comes to relationships, so working with a model of interconnected entities makes the most sense for using graph databases. In order to import the data contained in tables as described above, the first step is to model the use case.
Finding a suitable model depends on the task. An overall suitable model would be a graph capable of storing different node types and multiple relationships between entities. It’s called a heterogeneous graph.

It is worth noting that multiple types of graphs are used for data storage. RDF graphs, in short, store nodes and relationships as entities, while property graphs allow them to have properties, so a single node can contain multiple data values and types. We will be focusing on property graphs such as Memgraph and Neo4j.

But let's take it step by step. To construct a graph, we should understand the data in hand, what we want to model as nodes, and what to connect with relationships. Fortunately, table data is also modeled with entities in mind.

We can start with incidents, which in this case, have a specific table. We want to model this by a node type, so we can model each row of the `Incident` table as a single node, labeled as `:Incident`, to identify that node as containing information about an incident. Then, we want to know who was involved in an incident and have personal information about individuals who reported the incident by making an

insurance claim. So we can label nodes as `:Individual` and have them contain other information as node properties.

If you follow good practices while modeling tabular data, it shouldn't be a problem to translate them into graphs. In our example, most relationships represent foreign keys in tables. One exception is the connection between an individual and an incident, which we made from the `incident_individual` table.

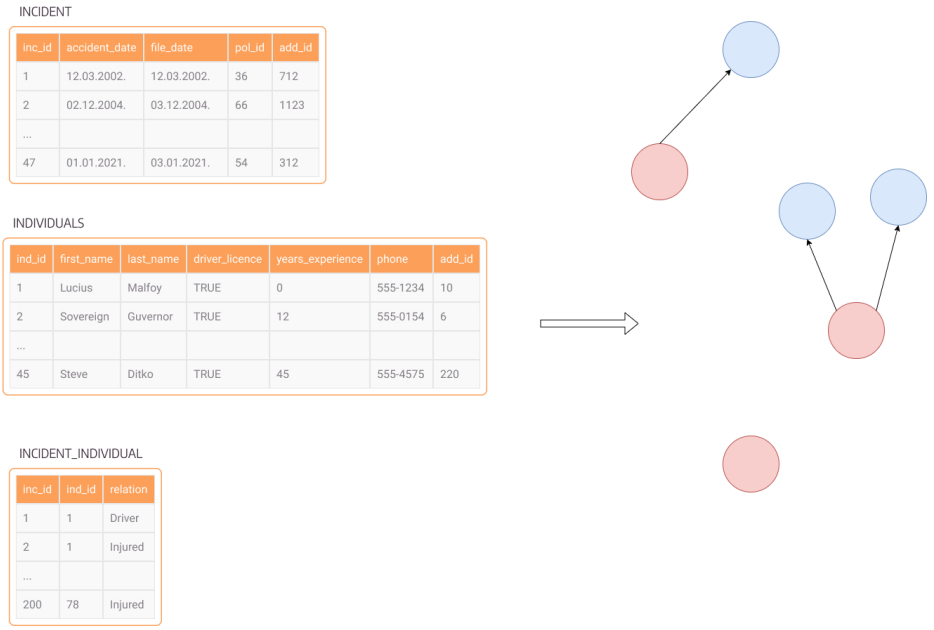

Figure 7. An animation showing tables turning into graph nodes, first incident, then individual, then incident_individual turning into relationships

In an ideal case, table rows are generally modeled as nodes, and foreign keys in those rows are regarded as relationships between nodes. In the case of associative tables, model many-to-many relationships for foreign key pairs. Row columns become node properties, and because graphs can also have properties on relationships, a relationship connecting an individual and an insurance policy can have a property that states the policy's expiration date. Upon expiration of the policy, the relationship can be removed with a simple query.

If you notice that many nodes connect to many other nodes, you are looking at a super-node. They can cause problems in the future, so it's best to avoid them by transferring super-nodes into a property. For example, rather than having a node "Policy Type", it is better to have a property named "type" in a node labeled "Policy".

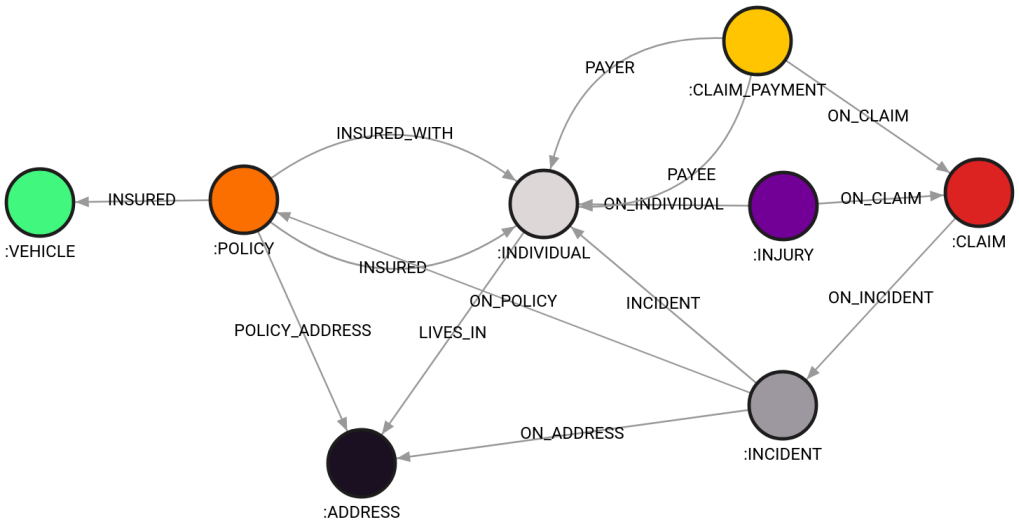

Figure 8. complete graph schema for insurance data

Furthermore, when looking at the data from Figure 1, think about how entities could be related in the insurance world. For example, they can share an address, meaning that individuals living at the same address are connected with many of the same nodes. Another example is payment, a unidirectional event involving the payer and payee. Information about those individuals and their relationships can be used when investigating events of interest.

Unfortunately, there is no cookbook for every use case. But on the other hand, creating a model without your hands being tied allows for more freedom and flexibility when working on specific tasks. Once your data model is figured out, it's time to start importing data.

Embark on the Fraud Detection Journey by Importing Data Into Memgrah With Python

With the fast evolution of data analytics and management solutions, insurance companies still largely rely on relational databases. To inspect insurance claims and utilize graph algorithms for systems like insurance policy recommendation and fraud detection, tabular data must first be imported into a graph database. We can do it

quickly and much more easily with Python without Cypher commands. Let's dig in to see how.

Move beyond Cypher when importing tabular data

The dataset we will use to showcase importing data into Memgraph, based on the designed data model, is insurance-based tabular data. It essentially models relational tables for insurance claims, policies, incidents, and other insurance-related processes. The tables are presented in the image below, along with attributes. The primary purpose of the data is to keep track of insurance claims along with all entities involved, their insurance policies, and payments.

To make the best use of the power of graphs, developers with existing tabular data need a way to import data into Memgraph.

The default importing process looks something like this. There would be a CSV file for every table, and you would need to load it and make a Cypher query defining how to create graph entities. Of course, Memgraph supports this with the LOAD CSV clause (https://memgraph.com/docs/memgraph/import-data/load-csv-clause). However, that assumes an understanding of Cypher, which is not always the case. Welcome GQLAlchemy.

GQLAlchemy (https://memgraph.com/docs/gqlalchemy) is Memgraph's fully open-source Python library and Object Graph Mapper (OGM) - a link between graph database objects and Python objects. GQLAlchemy supports transforming any table-like data into a graph. Currently, it supports reading CSV, Parquet, ORC, and IPC/Feather/Arrow file formats. All you need to do is create a configuration YAML file or object. Configuration YAML contains a definition of how a table transforms into a node, how entities connect in one table, and how multiple objects are cross-connected in tables created only for such connections.

One of GQLAlchemy's (https://memgraph.com/gqlalchemy) capabilities is to import tabular data from a relational database into Memgraph. You only need to write a few lines of code and define how the data should be interconnected. The code defines where the data should be imported from and allows more control when creating relationships between nodes, especially the many-to-many relationships. Let's see an example of how it's done.

Defining a transformation of tabular data into a graph with Python

Here is an example of a configuration file that shows how tables CLAIM, INCIDENT, and POLICY from Figure 1 would be transformed into graphs:

```yaml
indices:              # indices to be created for each file
  claim:              # name of the table containing claims with clm_id
  - clm_id
  incident:
  - inc_id
  policy:
  - pol_id

name_mappings:              # how we want to name node labels
  policy:
    label: POLICY           # nodes from the policy table will have POLICY label
  incident:
    label: INCIDENT
  claim:
    label: CLAIM

one_to_many_relations:
  policy: []        # currently needed, leave [] if there are no relations to
define

  claim:
  - foreign_key:            # foreign key used for mapping
      column_name: inc_id         # specifies its column
      reference_table: incident   # table name from which the foreign key is taken
      reference_key: inc_id             # column name in reference table from
which the foreign key is taken
    label: ON_INCIDENT            # label applied to the relationship created

  incident:
  - foreign_key:
      column_name: pol_id
      reference_table: policy
      reference_key: pol_id
    label: ON_POLICY
```

Once you get familiar with this template, adding another table is simple. For instance, to add an INDIVIDUAL table, you simply add its index to the `indices` field, and the label name to `name_mappings`.

Importing data into Memgraph

All that's left to define is a quick script that will do the hard work of reading the configuration file and moving the data from a tabular file format into Memgraph. The `ParquetLocalFileSystemImporter` can be swapped with importers for different file types and systems.

Currently, GQLAlchemy also supports connecting to and reading from Amazon S3 and Azure Blob file systems. See more about it in the table-to-graph importer how-to guide (https://memgraph.com/docs/gqlalchemy/how-to-guides/table-to-graph-importer).

```python
import gqlalchemy
from gqlalchemy.loaders import ParquetLocalFileSystemImporter

PATH_TO_CONFIG_YAML = "./config.yml"

with Path(PATH_TO_CONFIG_YAML).open("r") as f_:
    data_configuration = yaml.safe_load(f_)

translator = ParquetLocalFileSystemImporter(
    path="./dataset/data/",
    data_configuration=data_configuration
)
translator.translate(drop_database_on_start=True)
```

Once the tabular data from Figure 1 has been transformed into graph data, you can create a schema like the one in Figure 9, which shows how the entire database looks in Memgraph.

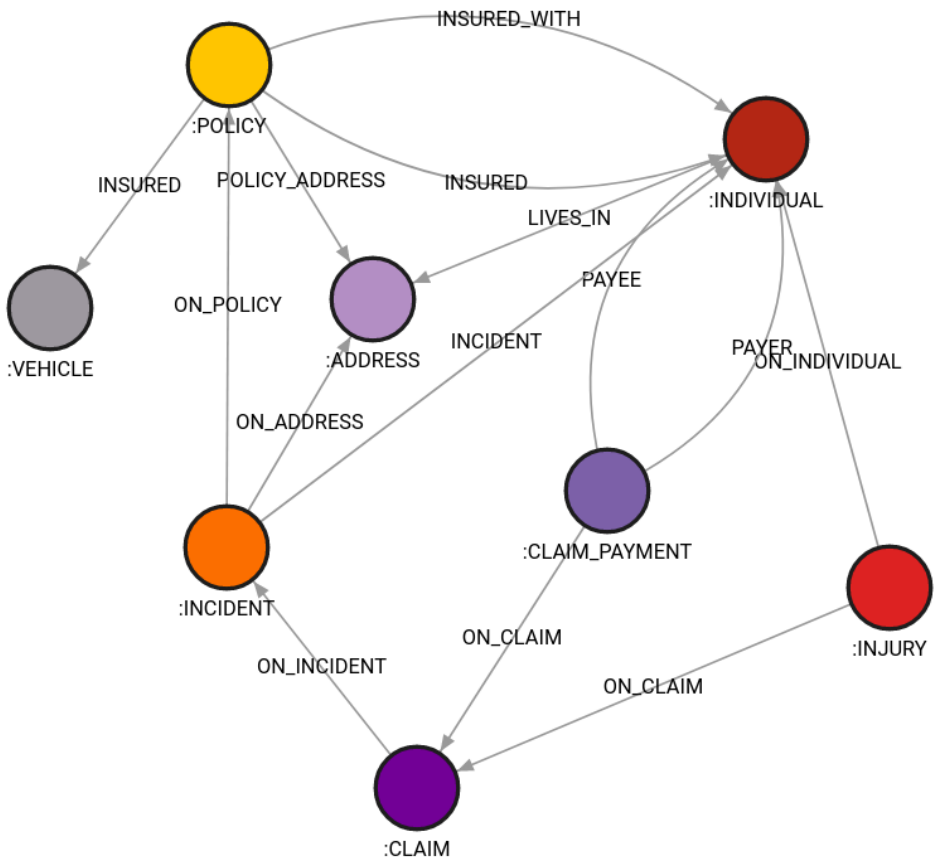

Figure 9. Graph data schema in Memgraph Lab

Try it out yourself before tackling your dataset and check out the insurance fraud Jupyter demo [https://github.com/memgraph/insurance-fraud]. In the demo, we generate a mock table insurance dataset, import it into Memgraph, and then use GQLAlchemy along with scikit-learn to design a machine learning system for fraud detection.

How to Visualize Connections in Insurance Data

We process information mostly visually, so it's always nice to have a schema or a representation somewhere when trying to understand concepts or ideas. The insurance industry handles various data regarding individuals with insurance policies,

incidents, vehicles, and claims. There are many benefits to organizing and storing this data in graph databases, and, using Memgraph, you can render graphs, look into claims, inspect suspicious actors, and more.

At a quick glance, we can see which nodes have many connections, like when an individual has many policies or was involved in many incidents in order to make certain recommendations, revisit contracts or detect fraud. On the other hand, we can see isolated incidents or visualize the size of payments made for insurance claims. We can also spot communities of fraudsters (fraud rings). Almost all graph data has some structure, and as nodes have labels, they make up a schema which is also a graph that shows us the structure of our data.

Memgraph Lab

Once you import your data into Memgraph, you can use a visualization tool Memgraph Lab to:

- Generate graph schema
- Explore and manipulate data
- Customize vizualizations

Generate graph schema

A graph schema of insurance data in a graph database is also a graph, as shown in Figure 8. It allows you to look into the structure of your data and better understand how entities are related. For example, if we want to expand our data model with information about the vehicle's lease or warranty in order to catch suspicious incidents occurring right before their expiration, we might want to look at the graph schema and determine how to incorporate this data into our model.

Lab also provides information about the percentage of entities (nodes and relationships) of a certain type with some property. Using that information, you can see how complete your dataset is, for example, if you are missing information on incidents that occurred. Possibly, if there are differences among a certain entity type, like traffic incidents and car theft, we might need different information on what happened, meaning we should probably split the incident nodes into two different types of nodes. Then you can model your data more easily by making a schema more understandable.

Explore insurance data and customize visualizations

The visualizations generated while exploring and manipulating data can help discover relationships much easier than comparing results in a table. By looking at visual query results run on insurance data, you can quickly deduct how many people were involved and what injuries occurred. You can then flag fraudulent claims and determine which relationships connect individuals to those claims.

Inside Lab, you can run Cypher queries and get results in both list and graph form. The graph view can be seen in Figure 9. By running this edge expansion query, you get all nodes within 3 hops of a chosen claim.

```
MATCH p=(claim:CLAIM {clm_id: ‘a6a1edd5’})-[e*..3]-(m)
RETURN *
LIMIT 100;
```

By customizing nodes and adding images to them, as well as highlighting the relationships of interest, you can get a clear picture of involved insurance claims and parties, as well as the vehicles and injuries.

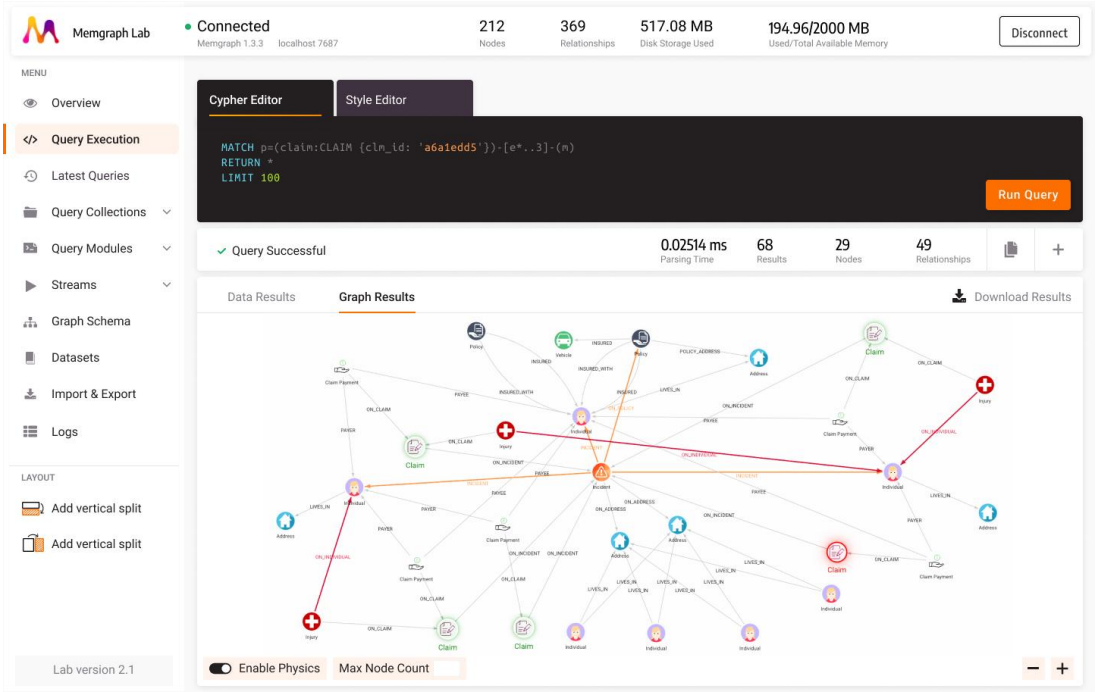

Figure 9. Visualization of insurance data with Memgraph Lab

By using graph algorithms, you can get more complex information from your data. For example, first, you run the community detection algorithm in Memgraph to assign a community to each node:

```
CALL community_detection.get()
YIELD node, community_id
SET node.community = community_id
RETURN *;
```

Community detection algorithms look for groupings of nodes based on how connected they are to each other. Then, run a query showing the communities of fraudulent claims and other claims are associated with them so that you can see if there is anything suspicious going on there:

```
MATCH (n:CLAIM {fraud: true}), (m)
WHERE n.community = m.community
OPTIONAL MATCH (m)-[e]-(a)
WHERE m.community = a.community
RETURN *
```

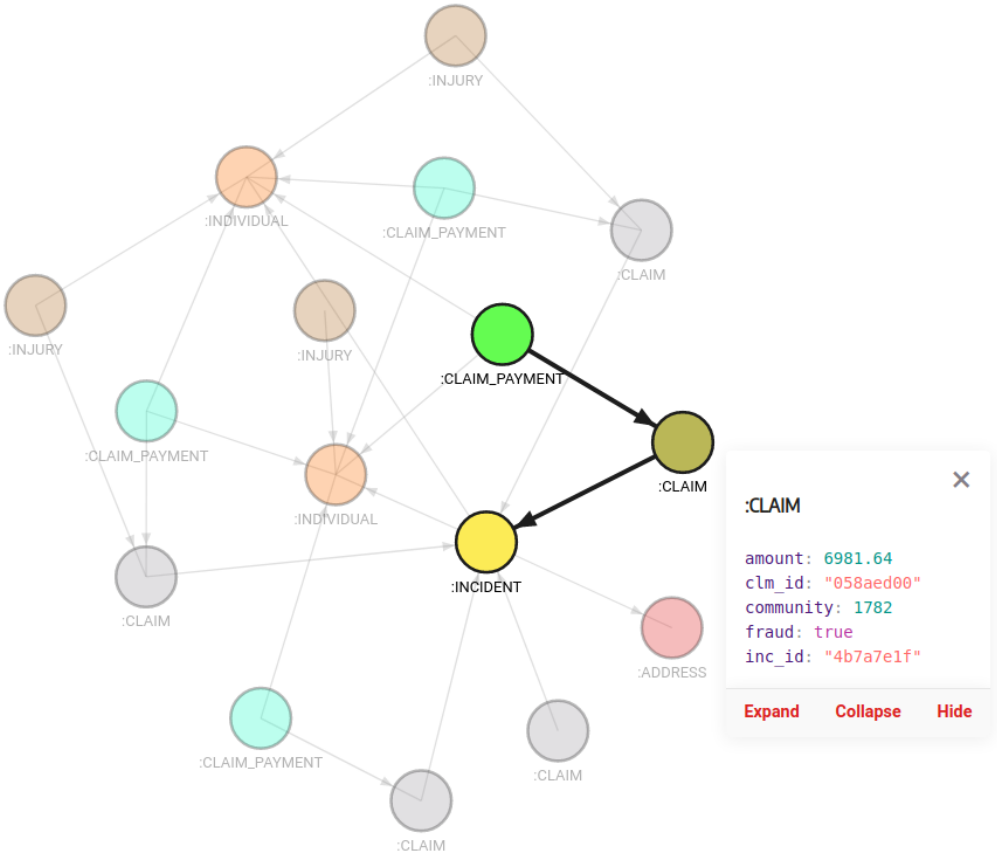

Figure 10. The fraudulent claim involved in an incident and its community containing other, possibly suspicious, claims

Another algorithm you can run to gauge an influence of a node (claim, individual) is PageRank, the famous algorithm used by Google to rank web pages in their search results. As it is available in our MAGE (https://memgraph.com/mage) library of graph algorithms, call it with the following query:

```
CALL pagerank_online.set(100, 0.2) YIELD node, rank
SET node.influence = rank;
```

Now, set the node size in the graph view to be proportional to its rank, or influence, set by the PageRank algorithm:

```
@NodeStyle {
```

```
  size: Sqrt(Mul(Property(node, "influence"), 200000))
```

The `influence` property is multiplied to scale, then the square root of the value is calculated to smooth the difference between very influential and less influential nodes. The query below will generate results that show the individual connected to two different incidents is more relevant than other nodes, as shown in Figure 11.

```
MATCH (n:INCIDENT)
OPTIONAL MATCH p=(n:INCIDENT)-[e]-()
RETURN *;
```

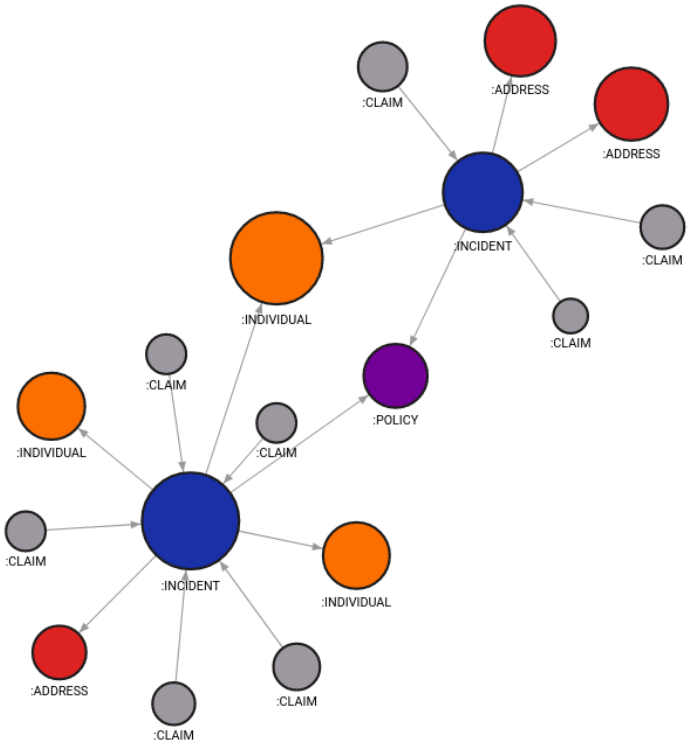

Figure 11. Incidents and their connected nodes

As shown with these examples, visualizations can be customized using the Graph Style Scrip by changing many aspects of nodes and relationships. By applying certain styles, you can highlight fraudulent claims or their connections to certain individuals and pinpoint extremely large claims that might prove suspicious.

Another feature Memgraph Lab offers is the automatic recognition of geographical data such as latitude and longitude properties. The data can be shown on a map, like in the heavily stylized example in Figure 12.

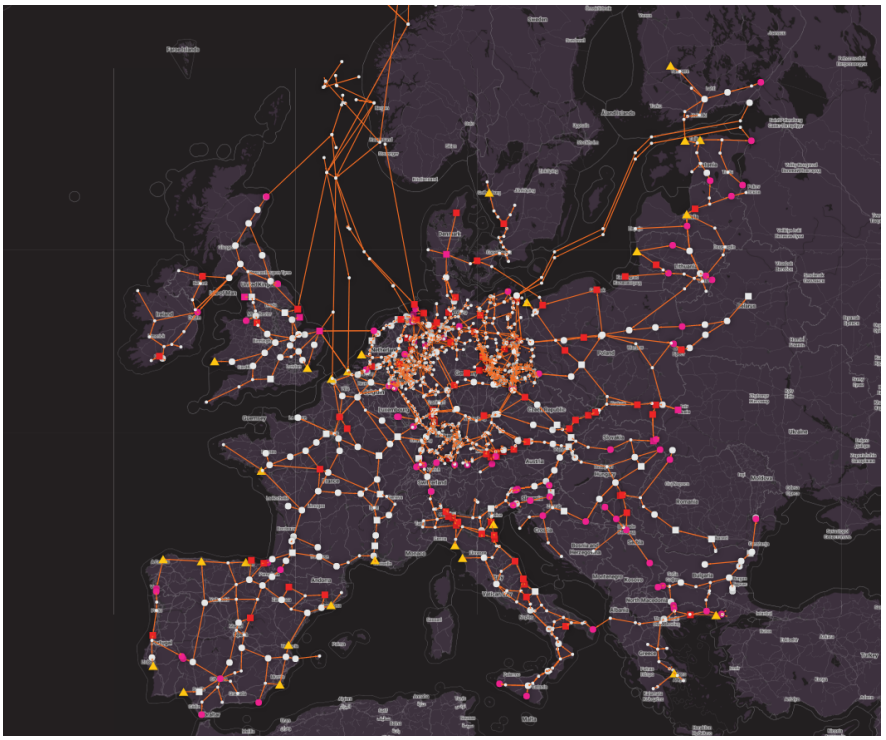

Figure 12. An example of a geographical graph

This feature can be used with insurance data to map out the locations of incidents. If certain locations prove to be frequently stated in insurance claims, you can investigate further as to why the accidents happen precisely at that spot and possibly discover an elaborate ruse.

Orb data visualization library

This high-level visualization is enabled by Memgraph's library Orb, a visualization engine for graphs. It can be used in a browser environment by [importing a script](https://unpkg.com/@memgraph/orb@0.1.4/dist/browser/orb.js), or in a JavaScript/TypeScript project through `npm install`.
Once you try out Lab, you will be familiar with the look and feel of Orb. You can use it to make custom visualizations and integrate the rendering engine into any internal product or app you might use for handling and analyzing insurance data, for example, you can use it to create a dashboard for insurance agents to track and inspect incidents.

Orb can also track events such as node/edge clicked, render started etc. You can subscribe to those events and do any action on top of them. You can use this feature with a tool that will help agents quickly go through many insurance claims and by interacting with the graph mark claims that have been looked into or have been suspected of being fraudulent. You could also make the interface interactive by allowing agents to create nodes and edges without doing Cypher queries.

Orb also supports physics simulations to change the graph's structure dynamically, thus creating visualizations you need to make your data show exactly what it represents.

If you are interested in Orb, check how to customize physics simulations, events, styling, etc.

Create Hybrid Machine Learning Model for Fraud Detection By Using Graph Data

Insurance companies base their business on managing risk. Their clients are rational and act reasonably and the insurance company has that assumption baked into their risk calculation.
However, as in any other business, sometimes clients act against the company by constructing various imaginary scenarios backed up with pieces of fake evidence. They still report a claim and extract cash from the incidents that never happened in reality if it passes the checks. The insurance company is at a loss as it did not detect the fraudulent claim. That is a problem. Luckily, it is a solvable one.
To detect fraud, insurance companies create modern solutions based on artificial intelligence to identify such scenarios and reduce losses. This one is for you if you know how machine learning works and have a fraud detection problem.

The right tool for the job

In insurance data, fraud is an anomaly. There are a number of possible ways to detect anomalies. One possible course of action is to create a rule-based system that will be triggered by an event similar to an event that already exists in the database and has been proven fraudulent. For example, if an incident occurs at the same address the injured individual lives in (at home), and the claim is made several days after the

incident, this may be a reason to look into that claim. If it is found that the same lawyer is hired as in another fraudulent claim, the rule might be triggered and flag the claim for inspection.

Unfortunately, this approach might not work for many fraud scenarios because fraudster behavior changes over time. Fraudsters are cunning, and these events are so unique and different from one another that the rule-based system simply cannot register, and they slip.

One adaptable, novel solution is applying machine learning. Nowadays, machine learning provides solutions to many complex problems. It can also be a powerful tool in fighting fraud if applied correctly.

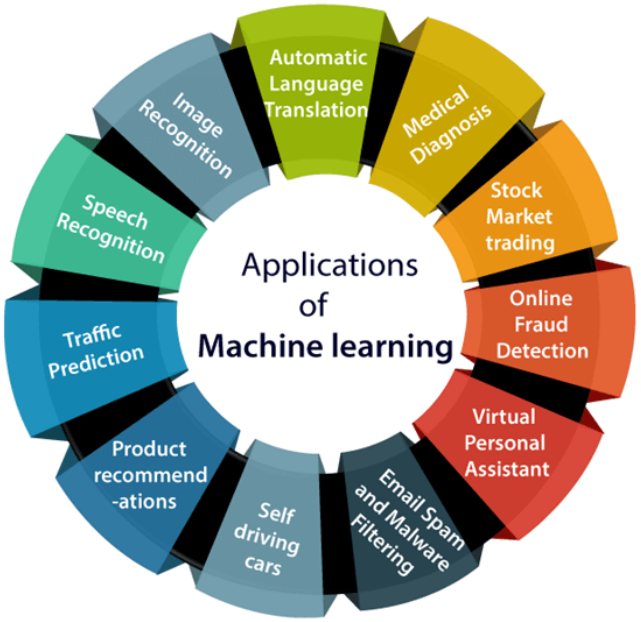

Figure 13. Machine learning applications
(https://www.javatpoint.com/applications-of-machine-learning)

There are various ways how to apply machine learning for fraud detection in insurance. Let's start with the basics. Machine learning involves a mathematical model, which, when applied to data, can solve tasks and process information on a level no human could ever do. The mathematical model can start as simple regression to a complex system of linear matrices and finish as deep learning.

The mathematical model is adapted to certain needs by applying data. To generalize better, machine learning models are given large amounts of data to learn from and adapt the model accordingly. Most ML models undisputedly benefit from the amount of data they have ingested: the more data you have, the "smarter" the model gets.

Insurance is an eventful industry, and the amount of data generated daily snowballs. Even though the data is plenty, not all data can be used to train the model as the data needs to be time-dependent, balanced, and relevant.

To achieve time dependency, we decide upon a specific “current” moment in the data used for training. All the data with a timestamp before that moment is used for training, whereas the rest of the “future” data is used for testing. Based on the testing data, the model will make predictions and test its correctness. This training process is represented in Figure 14.

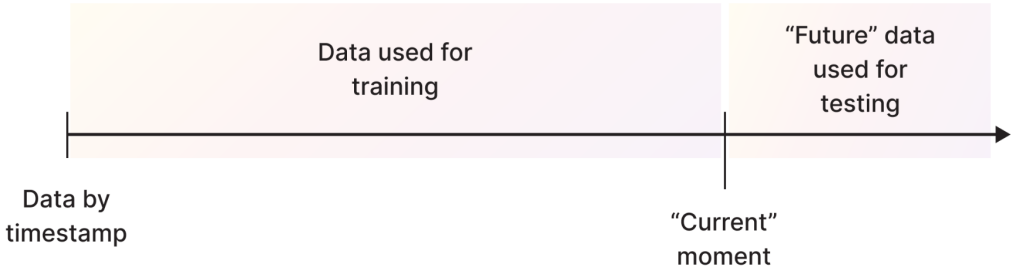

Figure 14. Training split

To make the data balanced, we need to understand what makes it unbalanced. In reality, fraud makes up a small but costly percentage of claims. When training a machine learning model, this might be a problem. If the model learns from too many negative examples, it might never be able to adapt to flag positive ones. There are multiple ways to resolve this. The most common is to upsample the fraudulent data to some reasonable extent, so it ends up more often in the model’s parameter adjusting process.

And lastly, we need to make sure the data is relevant. This includes deciding what kind of features the model should ingest. What is the right set of information that will enable the model to predict fraud on unseen data successfully? Well, let’s start by applying common sense to the problem. One idea is to include all relevant information someone would include when building a fraud detection system.

Important factors in fraud detection are the duration of the policy, policy premium mode, policy expiration date, whether the involved individual has a criminal record, how many years of driving experience the driver has, and the amount of money involved in the claim. All these can be boiled down to numbers. Numbers that can be used as input to predict fraud.

And although these factors are sometimes sufficient to classify the event, some fraudsters still go unnoticed for a long time. As experts in the graph industry, we experienced the benefits of using graphs as a tool to enrich the features with hidden connections seen only in networks to improve the flag rate of fraudulent claims.

Season it with graphs

As already said, insurance works with connections between all the entities involved. With such an intertwined data network, previously hidden data can be obtained from graphs. One possible solution is to use popular techniques like Graph Neural Networks (GNNs). Even though they are quite effective, they are complex to maintain. To keep the simplicity of training, we propose the idea of enriching the machine learning model with additional features coming from the graph.

Graphs can find connections that would stay hidden if working with tabular data only. They also allow powerful analytics that unravels communities, calculate importance, and generate artificial embeddings based on the graph structure (similar structures have similar embeddings, or close structures have similar embeddings). Conversely, graph algorithms such as community detection, PageRank, or Node2Vec can gather information that wouldn't be possible with a human agent exploring the data. This is the power of graph-level features that tend to thrive in an environment full of interactions.

But because the standard factors in fraud detection are expressed in numbers, the features and connections obtained by graphs also need to be expressed in numbers. Below, we separate insurance claims with community detection (red and yellow communities) and use the information on how many fraudulent claims there are in the blue node's community as a feature that is then used in training the model. Teaching the model that frauds can be connected at some point (fraudsters used the same lawyer, incidents happened at the same address or were committed by the same person, etc.) can help shape prediction, especially if the incident is already leaning towards being fraudulent.

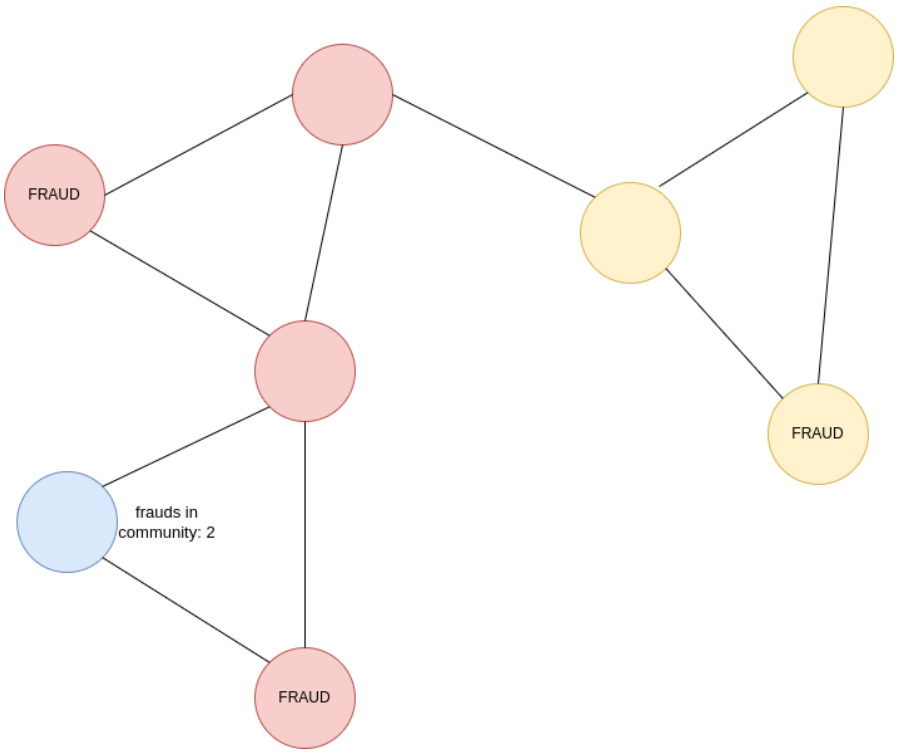

Figure 15. How Community detection might be used as a feature

Going hybrid

By combining graph algorithms and machine learning, we are using the best of both worlds and can create a powerful solution. The final machine model for fraud prediction should use both human-generated features and the ones obtained from graphs. This is the perfect solution for insurance cases where graph features play an important role in the final prediction by exploiting previously unseen frauds.

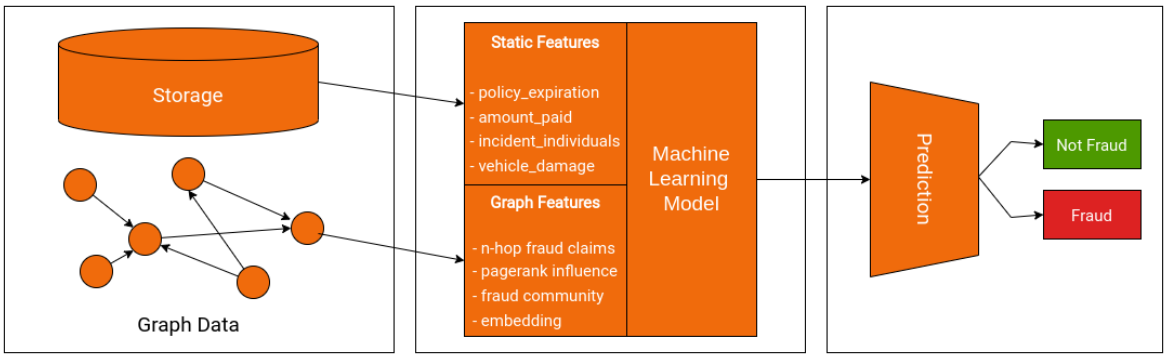

Figure 16. Model Overview

Because of the practicalities and legal aspects involved with unveiling fraud, the model's results need to be explainable. Machine learning models are not always the best at being explainable. For an interconnected, layered neural network, it is not easy (if at all possible) to trace the output decision that the network has made to its inputs in a way that we could point to inputs that play greater parts in the insurance claim being flagged as fraudulent. But some machine learning models are explainable, and simpler models are easier to interpret.

One approach could be using a decision tree model. This model enables full explainability,
where the decision can be formulated as a list of conditions that led to that decision. We can always follow the tree branch that shaped the decision and infer which inputs contributed to which degree to the model's decision.

Another approach would be a simple logistic regression. In logistic regression, parameters are trained to weigh the proportion of importance for some feature. Compared to linear models, logistic regression gives us a probability evaluation of its output, and with it being a simple model, we can more easily explain to which degree some inputs point to a fraudulent claim.

Lastly, experiments showed that using multiple models for specific cases can lead to better results. For instance, theft cases and related activities differ from car crash incidents. Within various specific cases, multiple models can be trained to prevent bias. The theory states that a low-variance model can be obtained by having multiple high-bias models simultaneously. This enables each model to be trained for its use case and used together with others to create a well-performing fraud detector.

Conclusion

In the insurance industry, data is key, and millions of insurance claims and incidents make up a type of social interaction between themselves. Graphs are ideal for modeling such networks and can substantially reduce storing and representing data complexity. Once the data is imported into a graph database, you can create hybrid machine-learning models to improve fraud detection in your system and save millions. Using graph expertise, we can augment the machine learning model with features gathered from graph algorithms, thus gaining better predictions and more insight into the data. Also, using explainable or simple models such as decision trees and logistic regression can give insurance agents information about where to look for signs of fraud.
When it comes to visualization, Memgraph took a lot of time to perfect its visualization tools - Memgraph Lab and Orb library. Once the data is modeled and imported into Memgraph, run powerful analytics such as machine learning to generate graph results. Then, customize those results to efficiently gain insight into the way humans are built to learn - using our visual senses.

Another Memgraph product is GQLAlchemy, which enables you to import existing tabular data into graph form with Python. Dealing with objects you are already familiar with can ease the stress of a strenuous job such as import and let you concentrate on the job to come - to make your company tap into the power of graphs.