Temporal Graph Visualizer

Authors: Diego Orlando, Joaquin Ormachea
Co-directors: Alejandro Vaisman, Ariel Aizemberg

2020-11-17

Abstract. Real world scenarios are increasingly being represented with graph databases. Networks in general, and social networks in particular, can be represented as node in a graph, linked through edges. When relationships and node include temporal information, the graphs are called temporal. Temporal graphs are then, graphs that keep track of the history of their nodes and edges. Although these scenarios are normally found in real-world scenarios, there is no tool in the market that can handle appropriately the temporal dimension in graphs. The present work introduces a platform to address this problem. The framework presented here allows displaying temporal graphs and navigating them across time. The result of queries expressed in a high level temporal query language can also be captured and navigated using this tool.

Keywords: Graph Visualization, Temporal Graphs, Temporal Database, Neo4j

Contents

1 Introduction

These days, organizations are facing the need of not only managing large databases but also of being able to gain insight from them. This makes relationships between data points as relevant as the data points itself. Graph databases address this need. These databases follow the NoSQL paradigm, therefore they are schemaless, and represent and store data by means of nodes and edges, being ideal to produce information and discover implicit relationships between data. Furthermore, the use of graph databases also provides other interesting benefits over traditional databases. These are:

- Performance, since traditional structures must deal with the rapid increase in relationships depth.
- Flexibility, thanks to not needing to define a fixed schema and allowing to restructure data model along with application and business change.
- Agility, due to its perfect alignment with agile and test driven development of today.

While graph databases have been a solution to store and display highly connected information, there is an important dimension which is normally missing: time. Temporality is a key factor to consider since it helps to represent data and their relationships across time. For example, in a social network, where relationships change constantly, it is easy to find that two nodes that have been connected at some point in time, are now disconnected. So adding the temporal dimension to the relationships and nodes can correctly determine the validity of the queries.

As another example, assume a user is working with a graph database representing flights between airports. She wants to find a trip from New York to London. When querying for flights between these destinations, a relationship pops out as there are lots of flights between these two cities. But a user who wants to travel not only wants to know if there is a flight at some point in time, but also, when is that flight happening. By adding the time of the flights to the relationship, we are creating more relationships between these two cities, each one representing a different flight at a different time, which lets the user search for the time desired as well as the date, which results in a better experience.

Since temporal graphs store the state of the data and relationships across time, the amount of information that they contain is normally huge. Thus, a relevant problem that arises when working with temporal graphs is how to display these graphs (usually very large) in a way that can be useful and friendly to the users, regardless how data are stored. Due to the large amounts of information and the level of abstraction present in these databases, showing properly the history of a graph is not a trivial problem. As mentioned before, some relationships may be present at a given time and then disappear, so an accurate way to differentiate each relationship is needed.

Debrouvier et al [1] propose a model and a query language (T-GQL) to address temporal graphs, that is, to represent and query the evolution of a graph across time. They also present a high-level query language that allows the user to query Neo4j [1] databases that comply with the Temporal Property

[1] https://neo4j.com/

Graph model structure, providing mechanisms to leverage their interval fields in order to get the full potential out of the database.

T-GQL queries are executed through a client tool called TDBG. However, as in all temporal query languages, visualizing the results in a way that allows the user to navigate the history of the database, remains a challenge. In this project, we address this problem, and implement a framework for interaction and visualization of temporal graphs which will be addressed as Temporal Graph Visualizer (TGV).

1.1 Contributions

This document describes the design and implementation of a web platform and tools that allows visualizing and querying temporal property graph databases using T-GQL query language. The key idea is building a web platform that provides abstraction from the structure that extends property graphs with temporal information.

More concretely, this work includes:

- A user interface allowing to define settings for connection and visualization characteristics.
- A query editor for the T-GQL language.
- A temporal graph visualization tool for proper representation of the database information.
- A filtering tool for results manipulation, either over information types or temporality for nodes and edges.

1.2 Document Organization

This document is organized as follows. Section 2 studies related work to understand the current context of the relevant topics, not only for visualization but also for temporal graph databases. Section 3 provides with a reflection of problems encountered during the development. Section 4 describes, documents and reviews the development process for the whole work. Section 5 explains how the implementation was done and every decision taken in order to achieve the proposed goals. Section 6 has an in-detail analysis for testing where the scope of the project is showed. Section 7 discusses limitations and finally Section 8 proposes further improvements for future development.

2 Related work

This section reviews existing work about general concepts in the world of information visualization, existing visualization tools, and the different temporal graph models that tackle the problem of temporality in graph visualization.

2.1 Information Visualization

Visualizing information in a way that it is both informative and appealing to read has been a research topic for many years. To begin with, it is important to define what information visualization really means. Stuart et al. [2]

define information visualization as the “use of computer-supported, interactive, visual representations of abstract data to amplify cognition”. Lima [3], a leading voice in the information visualization field, mentions three hypothesis that he comprises and explains:

1. Humans prefer curves: He says that from the moment that we are babies we show a preference from curves, a statement corroborated by Bar and Neta [4], which reveals a strong human preference for curved objects and typefaces. Also, in Chatterjee et al. [5], a similar inclination in architectural spaces was reported.

2. Circles equal happiness: This theory is explained by an experiment by Bassili in 1978 [6], where the faces of participants are painted black and subsequently are covered by dozens of luminescent dots. Participants are then asked to express different emotions in order to better understand the visual contour of each sentiment. Using Lima’s words, this experiment concludes that: “while expressions of anger showed acute downward V shapes (angled eyebrows, cheeks, and chin), expressions of happiness were conveyed by expansive, outward curved patterns (arched cheeks, eyes, and mouth). In other words, happy faces resembled an expansive circle, while angry faces resembled a downward triangle.”

3. Spherical geometry of the eye: In this third hypothesis he talks about how the circular framing and spherical geometry of our visual field, which causes a distortion similar to a "fish-eye lens" or a "crystal ball", could further "reinforce our innate tendency toward all things circular. Perhaps the brain prefers forms and contours that have a better fit within such a conditioned field of view.”

Another relevant work from Manuel Lima, is the "Information Visualization Manifesto" [7]. The manifesto explains that any information visualization project should follow 10 rules. From these ten, two are the most relevant to the present work. The first one is "Interactivity is key". About this he says: "By employing interactive techniques, users are able to properly investigate and reshape the layout in order to find appropriate answers to their questions. This capability becomes imperative as the degree of complexity of the portrayed system increases." The second rule is "Embrace time", from which he highlights that: "If we consider a social network, we can quickly realize that a snapshot in time would only tell us a bit of information about the community. On the other hand, if time had been properly measured and mapped, it would provide us with a much richer understanding of the changing dynamics of that social group. We should always consider time when our targeted system is affected by its progression."

Finally, it is important to mention a very well-known visualization tool called Observable.[2] One of the creators of this tool is Mike Bostock, another influential author on the subject of visualization, and key developer of the widely used Javascript library d3.js. Observable is a Javascript sandbox for code sharing and open collaboration in which non-specialized users can visualize data in real time, which uses d3.js as the visualization tool.

[2]https://observablehq.com/

2.2 Graph Visualization

Graph visualization is a particular branch of information visualization. The Visual Complexity platform,[3] is a unified resource space for anyone interested in the visualization of complex networks. It was launched in October, 2005, and its main goal is to leverage a critical understanding of different visualization methods, across a series of disciplines, as diverse as Biology, Social Networks or the World Wide Web. There is an interesting project about websites which can be visualized as graphs which was created in 2006 by Sala [8]. HTML consists of tags, like the A tag for links, IMG tag for images and so on. Since tags are nested into other tags, they are arranged in a hierarchical manner, and that hierarchy can be represented as a graph. By adding some processing, Sala creates an application that visualizes any input URL as a graph.

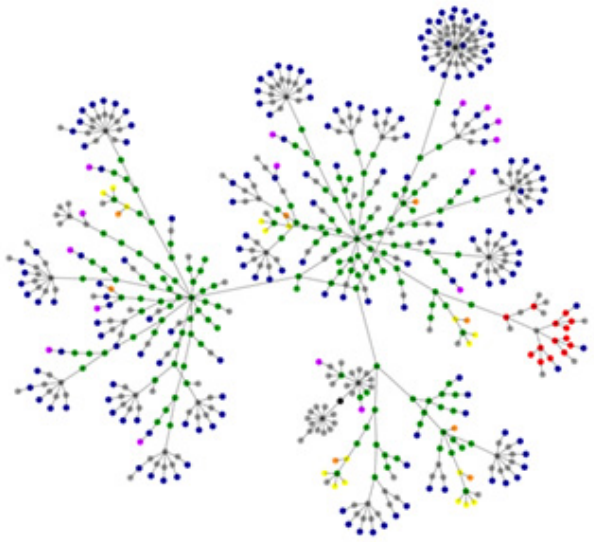

Figure 1: Graph visualization of the website msn.com by Sala.

Figures 1 and 2 show examples of websites visualized as graphs. 1 shows the MSN[4] website and 2 shows the Yahoo website[5]. Each node corresponds to a an HTML tag from the web page. Blue correponds to the the A tag, red for tables (TABLE, TR and TD), green for the DIV tag, violet for the IMG tag, yellow for forms, orange for line breaks, black for HTML tag, and grey for the rest.

Moreover, Bostock published a notebook which allows visualizing a temporal network that changes over time, using Observable. The work, called Temporal Force-Directed Graphs [9], shows data from face-to-face interactions at a two-day conference, where each node and link has a start and end specifying its existence. In Figure 3, in the top right left corner, there is a slider which shows the progression of the graph, with a play button and the date from which the data are obtained. A similar tool is used in the project presented in this document. The previously mentioned work introduces the concept of dynamic

[3]http://www.visualcomplexity.com/vc/
[4]https://www.msn.com/en-us
[5]https://yahoo.com/

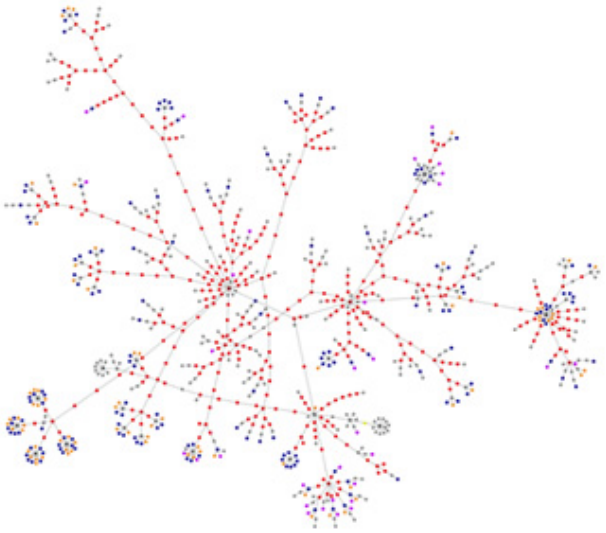

Figure 2: Graph visualization of the website yahoo.com by Sala.

graphs, showing that as information changes, visualization should change at par.

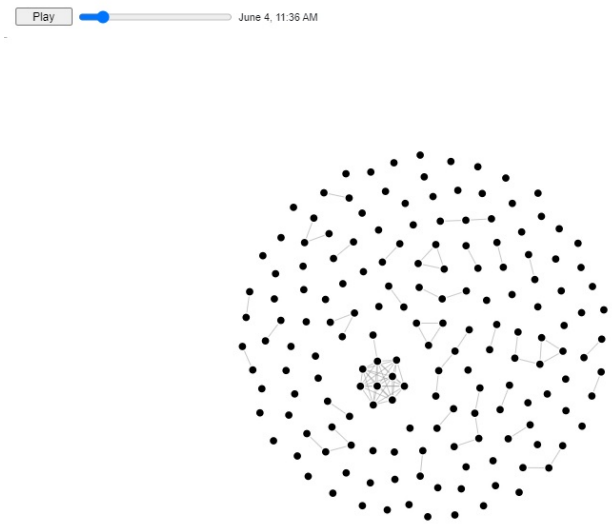

Figure 3: Temporal Force-Directed Graph by Mike Bostock.

NeoVis [10] is an open-source and publicly accessible graph visualization tool created by eleven developers from Neo4j. This tool is built on vis.js, a very popular Javascript library.[6] It works with Neo4j, the most popular database for graph visualization. It is easy to use, as it only needs a query in Cypher

[6]https://visjs.org/

language, and it returns an already built graph that can be shown directly on any canvas. It also allows the developer to tweak some features, like colours or families to generate different groups.

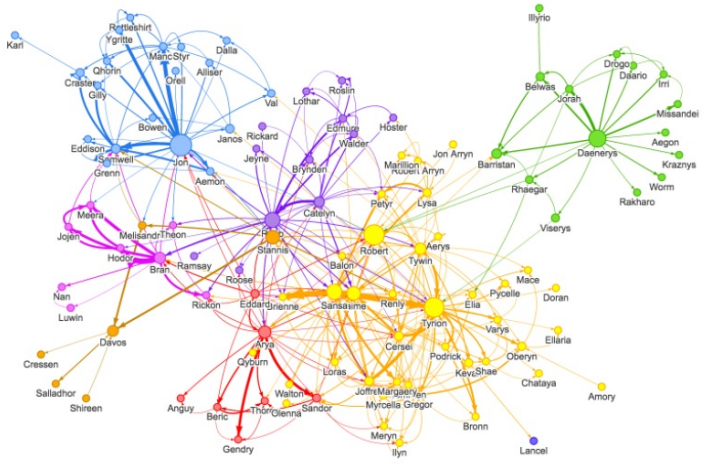

Figure 4: NeoVis visualization of a social network.

One of the most recent developments for graph visualization from Cambridge Intelligence, ReGraph [11], is a full plain graph visualization tool designed for React[7]. In 2019, ReGraph was released focusing all of their efforts in building a toolkit specific for React development. React is actually one of the most used frameworks nowadays for web development, and the one chosen in the present project to solve the dilemma of temporal graphs' visual representation. This API provides a number of fully-reactive, customizable components that can be embedded into applications. It has two visualization components: a chart and a time bar. To update, filter, style or highlight items in the data, users push an updated item's object into the component on the next render. ReGraph updates the React network visualization to reflect the change. Like other React components, ReGraph sits in the front end of the application, completely separate from the back end. Data are passed as a plain JavaScript object.

2.3 Temporal Graph Models

We classify temporal graphs as: duration-labeled, interval-labeled and snapshot-based.

2.3.1 Duration-labeled Temporal Graphs

These kinds of graphs are not property graphs, meaning nodes and edges are not labeled with a sequence of attribute-value pairs, but simple graphs where nodes are represented as strings, and the edges labeled with a value representing the duration of the relationship between the two nodes. The main use of this kind of temporal graphs is scheduling problems, where some sort of shortest

[7]https://reactjs.org/

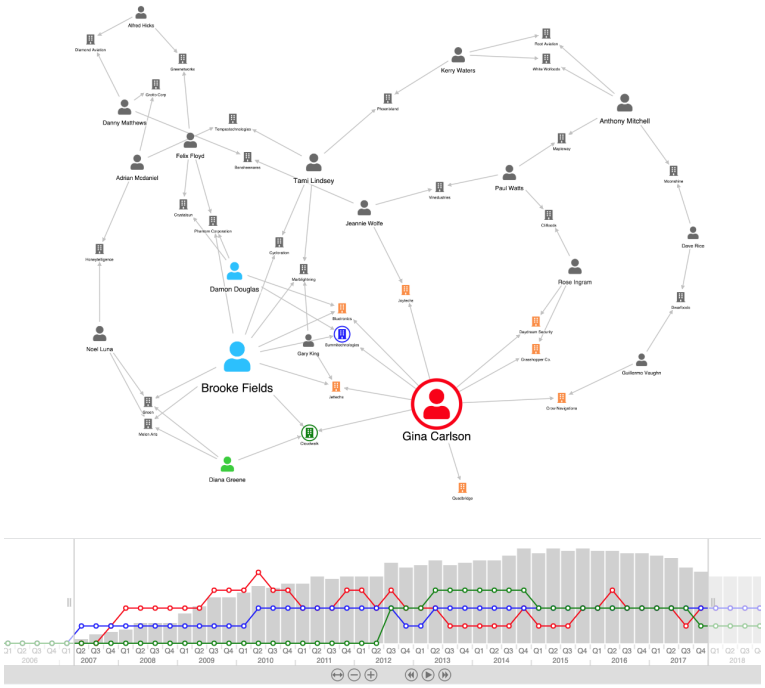

Figure 5: ReGraph visualization of a social network.

path must be computed, therefore, implementing some ad-hoc variation of the Dijkstra's algorithm.

In Debouvier et al. [1], the authors define four types of minimum temporal paths. These are: earliest-arrival path, which computes the earliest arrival time departing from a source location to a target location, latest-departure path, defined as the path that allows the latest departure form source location whilst arriving at destination at a targeted time frame, fastest path, where this path is the one with shortest elapsed time between source and destination and shortest path, showing the path of least overall traversal time from from origin to target location.

2.3.2 Interval-labeled Temporal Graphs

An interval-labeled temporal graph is a graph where each edge is a temporal edge representing a relationship from a vertex to another vertex, valid during a time interval and denoted as [ti,tf]. Valid time is considered in the remainder, that is, the times where the edges are valid in the real world, opposite to transaction time, which reflects the time where the information is stored in the database. This model also permits, for example, the representation of instant messaging where each label is time stamped with the interval [t,t] meaning that duration of the message is negligible.

The graph to the right of Figure 7 shows a graph equivalent to the one on the left, with the distinction of the type label used in the edges. In the graph on the right, for example, the edge between nodes b and g is labeled with the

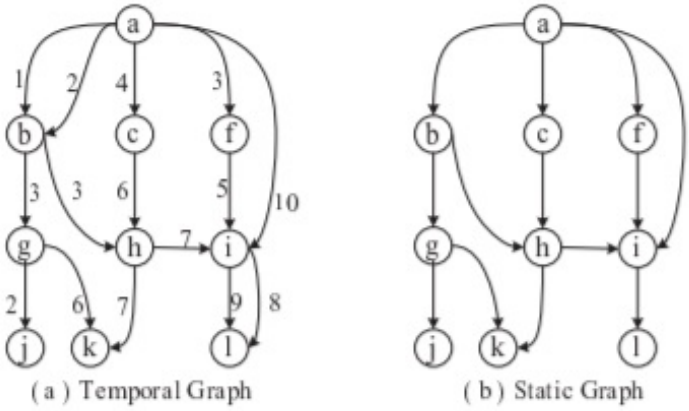

Figure 6: Temporal duration-labeled graph (a) vs static graph (b).

interval [3, 4]. This can be explained by comparing it to the same edge on the left graph, which is labeled with value 3, that represents the initial time, with a duration of 1.

This type of graph representation is ideal for social networks, travel schedules and trajectory representation, thus being the one chosen to develop throughout this document.

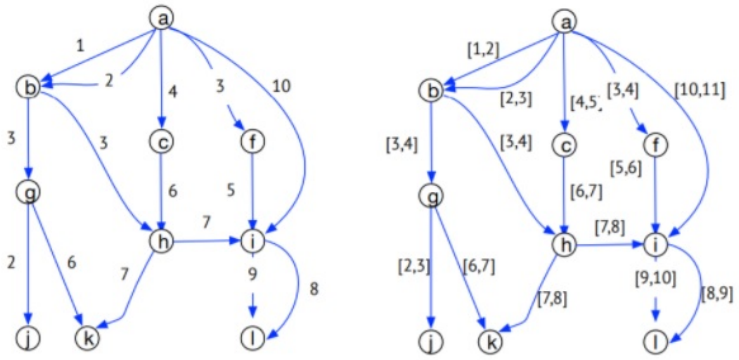

Figure 7: Left: duration-labeled temporal graph. Right: interval-labeled temporal graph.

2.3.3 Snapshot-based Temporal Graphs

Based on the work by Semertzidis and Pitoura [12], the history of a node-labeled graph is given in the form of graph snapshots corresponding to the state of the graph at different time instants. Given a query, it is a problem to efficiently find those matches in the graph history that persist over time, those matches that exist for the longest time, either contiguously (in consecutive graph snapshots) or collectively (the largest number of graph snapshots). These queries are called graph pattern queries. Locating durable matches in

the evolution of large graphs has many applications, like for example, lasting relationships in social networks.

Another relevant work is the one from Huo and Tsotras [13], which analyses the problem of efficiently computing shortest-paths on evolving social graphs. The authors use an extension of the Dijkstra's algorithm to achieve this for a time-point or a time-interval. Their main goal is to efficiently solve shortest-path queries within the graph's evolving history. This temporal queries are represented as a historical graph snapshot. For example, these queries in a social network can show how two people were connected in the past, with a historic snapshot of the graph at the time, and the evolution of their relationship is showed as updates.

3 Problem Description and Requirements

We tackle visualization of temporal property graphs using two different approaches. The first one focuses on the platform development and its functionality, and the second one focuses on the visualization capabilities that were needed to correctly handle the interaction with the database.

3.1 Platform

This work aims at providing a platform that encapsulates every functionality. Since the importance of providing intuitive interfaces is increasing [14], developing a comfortable and modern **interface** for user interaction is one our main goals.

According to Kreitzberg [15], users of software like this are usually thinking about the problem they are solving, so a well-designed one should reduce user's effort to think about the way of using itself to the minimum. So the idea for this work is to maintain a low cognitive load for the platform to allow users to focus on the data they are analyzing and also reduce the amount of training needed to use the application.

The platform presented here includes a space for users to write their own queries using the T-GQL query language. This work provides, then, a visualization platform that integrates with the query engine described in [1].

3.2 Visualization

We also address the visualization of a temporal graph across time, by selecting a date interval with a slider. **Time** is one of the most important factors to consider in the design of this project, as temporal property graphs are not static graphs, but they change in time. Surprisingly, there are not many proposals in this sense.

The property graph model [16] stores information with an underlying structure that allows to use nodes as property labels and values for them. The temporal graph data model adds structural information that allows handling time in a transparent way. This structure is described in [1], and the visualization tool must hide it from the user, leaving only the nodes and edges to be seen, since users are interested in looking at such information. To show them what they are really expecting we handle the **abstraction** of this underlying

structure when creating the visualization. This is described with further detail in Section 4.5.

Lastly, the tool to be developed must be able to handle a large number of nodes and edges, usually much more that the number of them than can fit on a screen. How to deal with these amount of nodes and edges is one of the challenges of this work.

4 A Platform for Graph Visualization

As stated before, one of the main goals of this work is to seamlessly integrate a graph visualization tool with the T-GQL engine in [1]. The first step set by this goal is to select the language and technologies.

4.1 Language and Frameworks

We chose Javascript as language, since it is a powerful and versatile language mainly used for front-end systems. The main drivers behind this decision are: facilitating our access to interface and visualization libraries, and bootstrapping our work allowing us to focus on the actual problems.

As part of the consequences of the previously mentioned decision, the selection of technologies and framework are encouraged to take advantage of Javascript libraries. React.JS framework was chosen in order to facilitate constructing the platform and handling user interaction. React.JS framework provides the work with structure and a thought model to organize the whole system.

4.2 Visualization Libraries

Javascript and React.JS decision, set some restriction when thinking about which libraries and tools should be chosen for visualization. Compatibility is no easy task, although it helps to shrink the options.

The compatibility constraint is due to the way the selected library and React.JS framework handle DOM elements[8]. React manages DOM elements through a Virtual DOM hooked to the Real DOM, so the chosen library must be able to adapt towards this. Failure to handle this will generate conflicts from React's side when trying to synchronize its Virtual DOM with the real one. Some of the most interesting visualization libraries and frameworks work with their own Virtual DOM or even modifying the real DOMin order to leverage visualization elements creation, which actually ends up messing with React's job.

In the meantime, while trying to solve this issue with some candidate libraries, advantages have been taken of these situation to explore most of the existing libraries. The resulting list from this research is too broad to go through so this section details only some of them. The most interesting ones that were not used are OGMA, G6, VX. Further discussion is addressed on Appendix A.

[8]According to W3.ORG [17] DOM is the acronym for Document Object Model and it is the application programming interface (API) for HTML and XML documents. It defines the logical structure by which the information of the document might be accessed or manipulated. Elements in this structure represents an object which encapsulates not only structure but identity and functionality. In an HTML web page, there is only one DOM that is visually represented and this document refers to it as the Real DOM.

OGMA	G6	GOjs
Popoto	Vis.js	ProtoVis
JSPlumb	VivaGraphJS	AmCharts
VX	React-Vis	

Table 1: Different Visualization Libraries Tested

Finally, **Vis.JS** [18] has been chosen because it is 100% compatible with React and there is a way to avoid component references and only re-render through normal components lifecycles. Furthermore, this library has an extremely detailed documentation and explains perfectly their solution to graph visualization. Regarding this last point, this is very compatible with the needs of this project according to the format and data handling. Distribution of the graph is not something that needs to be handled, so, just setting forces simulation the layout is automatically displayed. Also it provides a very well developed events API which is useful for interaction with the visualization.

Lastly, there is a React component wrapper which has already been implemented for their graph solution. This is perfect since having the main component already correctly embedded inside the react application as a reference is very helpful to avoid stepping into the problem of DOM management from each framework or library.

4.3 Platform interface

Regarding the application interface, the decision is to have a straightforward design where information is clearly presented to the user without any useless information generating distractions for them. In other words the intention is to have a **clean interface**. On the basis of having some material design styled user interface, the intention is to leverage the most out of other existing projects with open source licenses.

4.3.1 Layout

To fully provide the cleanness mentioned above, a simple layout is needed. Figure 8 depicts the welcome screen. This layout is the same for every view across the whole platform. The template theme that is used to bootstrap general component design is further explained in Appendix B.

Section 1 in Figure 8, is the navigation drawer which contains the different navigation links to switch between the different screens. This navigation drawer, as its name states, is collapsable into single icons in order to provide a larger main section area when preferred. This can be seen in Figure 9 . **Section 2** is the main panel and is intended to hold the varying information regarding the different tabs.

With this fixed layout the user is provided with consistency to be able to focus only on the information in the main section. This is similar to the way in which Bloom [9], the Neo4j's visualization platform, handles its interface and as most users will be handling Neo4j databases, it is a good idea to stick to what they are probably familiarized with.

[9] https://neo4j.com/product/bloom/

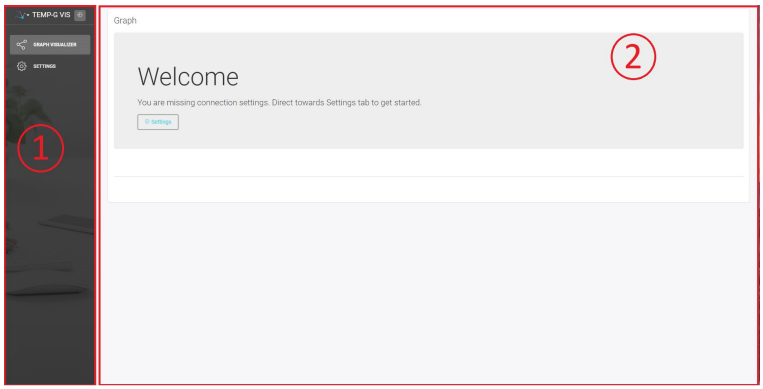

Figure 8: TGV noted layout

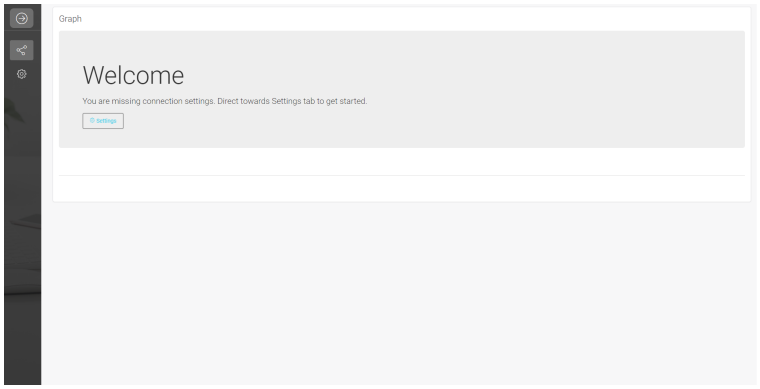

Figure 9: TGV collapsed drawer

4.3.2 Settings View

The first view is the **Settings View**. In Figure 10 the settings navigation item is selected on the navigation bar and the main panel has the different configuration settings available for the user. In order to comply with material design principles proposed in [19], the focus is on the layout and motion inside each view. As different information is needed from the user it is useful to group pieces of information into card modules appearing whenever they are required.

Beginning with just one module, the user can understand easily that to start configuring the application they just need to enter connection information. Once this information is provided, the user has only one button to continue and test the connection. The idea of motion and change from material design allows us to provide the user with a determined configuration flow.

Once the connection is successfully tested, the remaining card modules stack up under the connection setting's card. There are two distinguishable group settings, one for nodes and one for edges. Both are arranged on their own cards that contain a table with rows for each type. This allows an expandable layout that can easily adapt for the different possible databases.

All the information needed is retrieved through a direct query to the database that was provided in the previous module. Moreover, there is a default selector

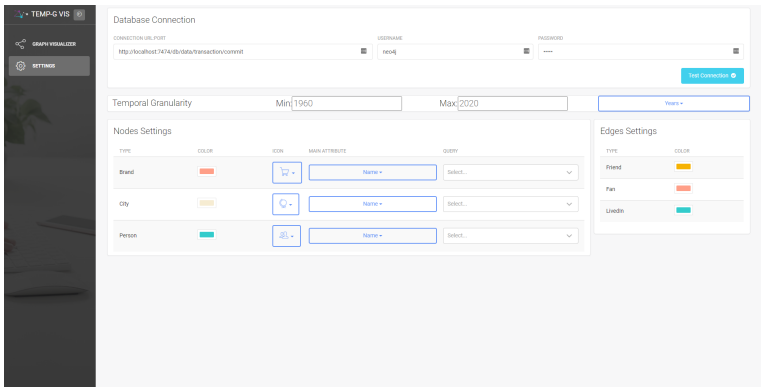

Figure 10: TGV Settings View

from which the user can choose to have the visualization view up and running as fast as possible.

Node settings

For each node type there are four available settings from where three are strictly used for visualization styling purposes and one has to do with the query generator. Each of this four columns has its own type of button. The most common ones are for the icon and main attribute settings with normal dropdown buttons.

- **Type**. Indicates the type of node corresponding to the configuration selected in the row.
- **Color**. This selector provides the user with the customization she wants for the visualization. Node color is is the main way to distinguish between node types among many nodes. The Color Picker has palette selection, gradient and even manual RGB color insertion. This is really useful for managing big graphs, since the user can easily relate similar colors with related node types and find anomalies in an instant.

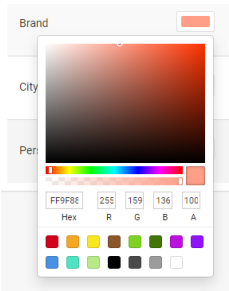

Figure 11: TGV Color Picker

- **Icon**. This option allows the user to choose from a list of pre-loaded icons to be used in the selection module on the visualization view to quickly identify the node type they are viewing.

- **Main Attribute**. This setting has two main purposes. Due to the nature of the temporal property graph structure, every object type can have several attribute nodes with their own value nodes. So a way to determine which of this variable options is more important to use for node distinction is needed. In the first place, this attribute is used to determine which variable is used to show while hovering over a node. Secondly, it is used to determine which attribute is wanted in lookup for the following column, related to the query builder.

- **Query**. Last but not least, there is a query filter builder. This feature allows starting focusing the visualization from the very beginning. The idea behind this selection is to determine if any of the node types should be filtered to just show a set of values. To fulfill this and prevent the user from doing unnecessary work, a specific component was defined. First of all, the component is populated through a query for the available values related to the selected attribute in the previous column. Secondly, selection inside the dropdown is through checkbox, to make it really easy to make multiple selections at the same time. And lastly, as lists could be quite big, the component also has a search box that filters the dropdown list through a fuzzy search [10]. This last feature not only makes the user find the correct values fast, but it also allows for some typo errors, providing with a list that has been already filtered with most close results.

Edge Settings

Regarding edges the user is provided with only one customization, color. As in every test case the only need for the user is to distinguish the type of the relationship, it makes sense that this is the only characteristic necessary. The selection component is the same Color Picker which has been described before for nodes.

Temporality Settings

As the document describes in the following section, the mechanism to manage and select temporality filters is done through a slider component. This slider component, that can be appreciated in Figure 13, has several characteristics that affect its usability depending on the type and amount of data that the user will be manipulating. To allow the user to work comfortably with the slider they are provided with options to tweak both limits and granularity.

- **Limits**. Since the data in relation to the visualization can be widely spread over time. To make things more clear, usually different node types like cities have temporal ranges that are in magnitudes (years) different that other ones, for example events (hours). So most of the time, the user queries about events that are related to cities, so the temporal spread of the whole result is fairly wide since cities influence this total spread. With the intention to automatize the limit this spread, the user can choose the limits of it, so that in cases of misleading general spread, the user would be able to use the slider for the filtering needed.

[10] A fuzzy search is done by means of a fuzzy matching program, which returns a list of results based on likely relevance even though search argument words and spellings may not exactly match. Exact and highly relevant matches appear near the top of the list. (see `https://whatis.techtarget.com/definition/fuzzy-search`). A comparison between the different available algorithms and their conflicts can be found at `https://aip.scitation.org/doi/10.1063/1.5114193`

- **Granularity**. Again, following functionality comments regarding the slider use, the best way to find precision is to define the step in which users can increment or decrement the selection for the sliding filter. Though, when granularity or step is too small the step is changed towards a continuous slider, desisting the idea of precision. This options provide the user with four step selections: Years, Months, Days, Hours. Depending on the spread of the slider, the selection transforms into continuous selection instead of discrete.

4.3.3 Visualization View

The second and main view is shown in Figure 12 as a screenshot of the main panel. In this view, following the same idea from previous views, several modules or boxes can be defined that clearly behave differently and group different characteristics.

This is intended to provide the user with the notion that they are working with a 'toolbox' where each tool is on its own place and allows them to use only what they need.

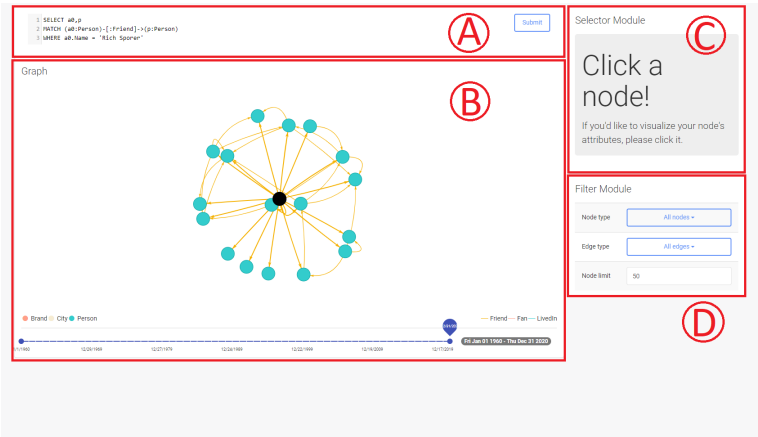

Figure 12: TGV visualization view

The different modules include the following:

- **A. Query Box.** On the upper part of the main panel, the root trigger for almost every interaction can be found. As it is described later in this document, the visualization tool permits customizing and exploring the results of temporal graph visualizations, so this box is used to input and edit the main query that can fetch the results from our colleagues service. This box component aims at simulating a code editor, and was chosen to provide consistency between this and the previous project, since they were using it. The name of the component's project is CodeMirror[11]

- **B. Graph Module.** This is is the main module where the results to the input query are represented visually. Figure 12 shows this module can be divided into two parts: the center one, which shows the visual

[11] More information related to the project can be found in https://codemirror.net/

representation, and the bottom part, with the color references to provide more information for better understanding of the visualization. Something important to highlight, that Figure 13 shows on the top, is the selection of the different shapes that conform the color references. In this case we leverage the use of shape and color as two variables that easily establish visual association allowing to use space more efficiently without creating confusion. Colors are used to distinguish types, while shapes distinguish elements, circles referring to nodes and lines referring to relationships.

This practice of taking advantage of n-dimensions of visualization comes from the idea of visual variables and their categorizations presented by Jacques Bertin on his work Sémiologie Graphique [20] originally published on 1968.

Figure 13: TGV Slider

Below these references, the slider which has two selection points to filter results for a period of time. As it was mentioned earlier, the granularity of the steps can be changed in the settings view and the extremes of the slider can be set manually as well. On the right hand side, there is a text that describes the period selected with the slider for clearer understanding of the filtering that is applied. This characteristic becomes important when using the continuous slider, since it is more difficult to precisely select the desired time.

- **C. Selector Module.** On the upper right of the panel there is a module to provide detail for any node that is selected. Since the nature of the data in Temporal Graphs allows nodes to have infinite information-related nodes with lots of values, it was decided to provide a separated box for exploring the content. Pushing this information into the visualization module can end up messing the neat representation and can interfere with the main objective of understanding the visual representation of the data. This module displays information about the node, referencing its color and icon for node type and listing its different attribute and value nodes related. Additionally, besides each of the attributes, the temporality related to them was added, so that behaviour can be better understood when filtering the graph through the slider.

- **D. Filters.** Last but not least, the filtering module on the bottom right section of the main panel. In this module, selections can be performed regarding the amount of nodes to be displayed and the types of nodes and edges for the visualization. Although this are simple filters which are performed after the query is consulted, they can be really handy to clean and focus on the visualization that really matters to the user.

4.4 Untangling the Information Flow

One of the main advantages for using react as a framework for the platform is its capability to naturally work with components and states. Since this web

application is intended to handle lots of interactions with the user and different data sources, this functionality becomes proves to be useful.

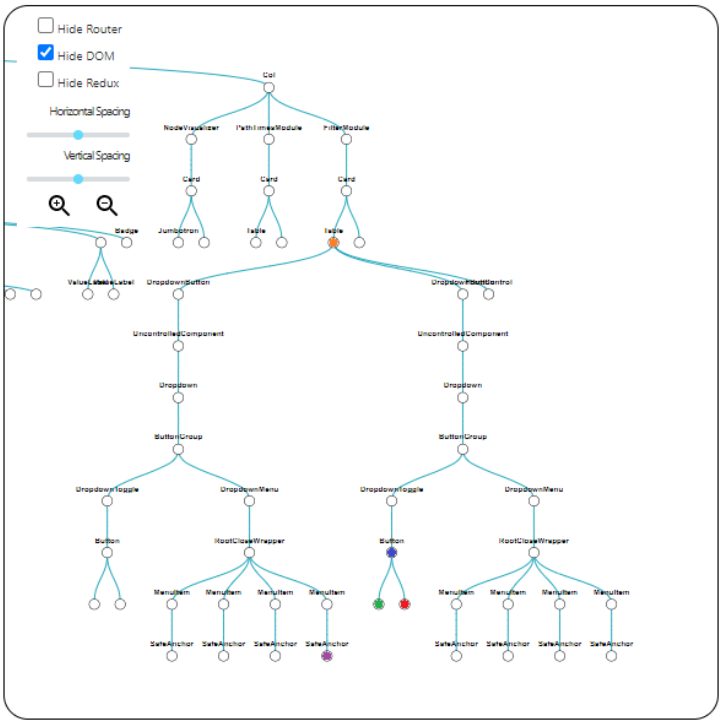

Figure 14: TGV's Component Tree Section

However, problems occur when components need to interact in a deeply nested tree from far apart branches in this hierarchy. Figure 14 is a segment of the react components' tree of the platform hiding the DOM elements. The right way of handling state is that each component is able to interact with its parent and its children, so when we are trying to get information from the red node towards the green node we have to go through their parent which is something quite easy since this is probably something doable in the same line of code.

The problem appears when these components need to interact between nodes that have a very far away common parent. This is the case for components red and violet seen in Figure 14. The only way to connect between them, without breaking the order in which information is intended to flow is through orange node, which is seven levels apart from each of them. Furthermore, this state must be stored in the shared component in order to be able to share this information in some cases without going all the way down towards the other side of the tree. So what it is usually done is to "Lift the state Up" allowing the information to bubble up towards the component in common. This is depicted in Figure 15.

This becomes quite problematic, since one has to waste lots of time developing wrappers to handle these methods and information is being passed through the different layers. So in this heavily component dependant applications another solution is usually brought to handle shared state.

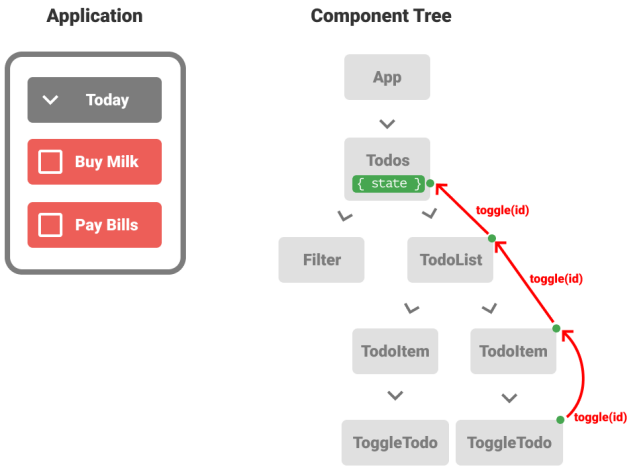

Figure 15: TGV's Component Tree Section

One popular solution would be to use Redux[12] but this one implies lots of overhead configuration for the few information that this application was holding in the "root" of the project. So the chosen solution was to use React's own workaround that was introduced with the feature of React hooks, Context[13]. The idea behind Context is to have a way to avoid this information flowing from the top component in the tree to handle data used by many different components at different nesting levels. It achieves this by creating Context objects which are hold in Providers. Then interested Components to subscribe to this Context and consume the broadcasted values through the information held in the nearest Provider.

In other words Context acts like a hub storing data widely used from elements across the whole platform and Providers are direct links towards this hubs. Then elements can interact with data in hubs as if data would be in an element right on top of them in the structure. This whole mechanism permits global information to be handled correctly without tangling up the information flow between components.

4.5 Underlying Structure Abstraction

One of the main goals of this project is to be able to abstract the underlying structure of temporal property graphs from user's handling. In Figure 16 the schema of the a social network temporal property graph is shown. This is what visualizations will usually show, since Cypher queries won't automatically hide this information from the user. In order to do so, users must explicitly build the expected outcome. A key feature of this work is that the visual interface hides these structural details.

[12]https://redux.js.org/
[13]https://reactjs.org/docs/context.html

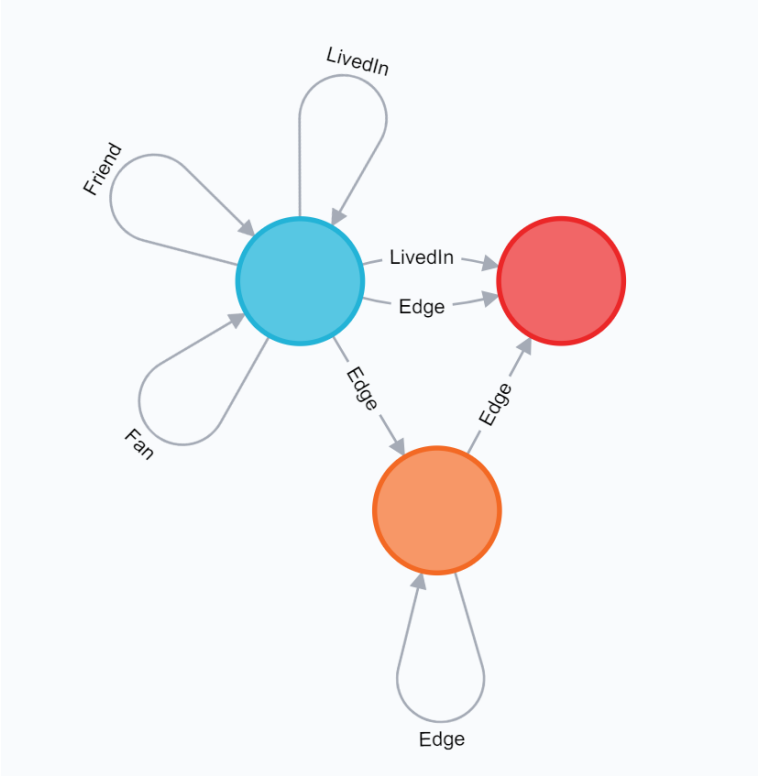

Figure 16: Social Network Temporal property graph example schema

Therefore, the project offers a way to visualize this same schema in a concise format, trying to shrink every attribute nodes and their corresponding values into object nodes, allowing user's to see the information only when needed by selecting that node but providing a cleaner look when consulting just for objects.

The other benefit that this work provides to the user is to be able to successfully visualize different object types as really different types through color representation. When we analyze the underlying structure we are only viewing all nodes as object. This can be seen in Figure 16.

In Figure 17 we can see this different object types and their relationships avoiding every other nodes that store their information, although this information is still accessible through other methods such as hovering or selecting the node. To the right of the same figure, we can appreciate one Person node being selected in order to explore values for its corresponding attributes without cluttering in the visualization.

4.6 Visualization Rebuilding Process

It is relevant to discuss the process through which the visualization of the graph is rebuilt. To properly describe this process, the steps of the normal work flow of the application are enumerated below:

1. **Connection Information.** The first step is to input the connection

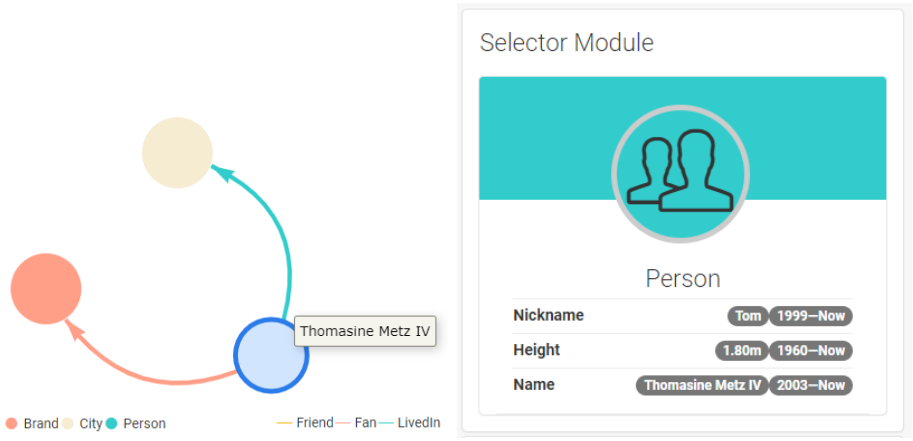

Figure 17: Relevant structure exposed and compacted attributes

information for the database. This immediately tries to use the credentials provided verifying that they are working and fetching useful information to fill the following section that configures settings that are specific to the database connected.

2. **Visual Settings.** Once connection is successful, user selects visual preferences which has been previously described in this document. Changes to this settings are instantly reflected on the visualization.

3. **Input Query.** Once user preferences are set, user heads towards visualization view and begins by entering the query corresponding to T-GQL language. This triggers the query towards TBDG service and later starts the **rebuilding process** mentioned above.

4. **Type Filtering.** After the visualization result is prompted into the main panel, user uses filtering through the filtering module to narrow types and amounts of nodes towards their interests.

5. **Temporal Filtering.** At the same time temporal filtering can be applied through the slider component, hiding every node and edge out of selected range.

6. **Node Selection and hovering.** Lastly, once the visualization is represented on the main panel, user is able to hover over nodes and edges to analyze information about types and temporal ranges at a glance. If more information is needed about the node, they are able to click on any represented node and the selection module fills with related information.

Back to the rebuilding process , the steps needed in order to take advantage of the TDBG service and the results it provides are described below. The following simple query example helps to understand the idea. It has been performed on TDGB service and on TGV:

```
SELECT p1.Name, p2
MATCH (p1:Person), (p2:Person)
WHERE p2.Name = 'Williams Little' and cPath((p1)-[:Friend*1..3]->(p2))
```

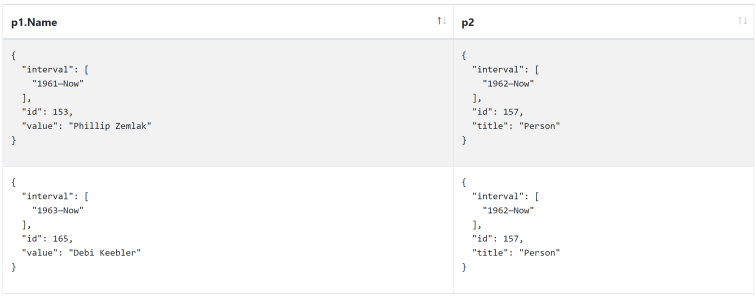

Figure 18: TGDB Service JSON response

- **Results from TDBG.** First, the response that the TDBG service provides is in JSON format. This includes the list of nodes and attributes that appear in the result to the query. As Figure 18 shows in an extract of the results table, the result provides no information about edges which are needed to be able to show them on the visualization.
- **Fetching attributes.** In order to be able to apply filtering and provide with hover information it is necessary to fetch attribute nodes and their values related to the nodes involved in the query result. Note on Figure 18 that when the user queries for the name attribute - in p1 node - the result comes with this attribute embedded on the node object, but in the case of querying only for the node, the attribute information won't be returning.
- **Fetching edges.** Performing a direct fetch to the database to get the missing information was proposed by our colleagues from the TDBG project. This is finally working by collecting every node id - [nodeIds] - from the result response and using it in the following query:

```
MATCH (n:Object)-[r]->(m:Object)
WHERE n.id in [nodeIds]
AND m.id in [nodeIds]
RETURN collect([[n.id,m.id], type(r), r.interval]
```

- **Differ from path response.** In the case of manipulating a path query response, a different mechanism is applied. The TDBG response gives the nodes involved in each path in the correct order. So each path is iterated and given a different path color code. The color code is stored in the node, thus when a node participates in more than one path, as each node can be painted in only one color, the last path in which that node appears is the one that gives the color to it. Each color code is an integer from 0 to 11. So the first path in the iteration has a color code of 0 and the last one a color code of 11. Color criteria is explained in the following subsection. Moreover, every different node encountered is stored in a list, that is used to fetch the edges involved with the mechanism which was explained above.
- **Build graph.** Using all these data, iteration through the result nodes is done, validating node limit and types from filters and skipping if necessary.

Once those filters are done, temporal filtering is applied by performing an interval check that is later explained in Section 5.1. Then, the color is chosen depending on whether the node is part of a path or not. If it is, the color which was previously chosen is used. If it is not part of a path, the specific color for that type of node is used (which was selected in the settings view). And before consolidating this node in the visualization, a check is made to corroborate if it belongs to the WHERE clause. After nodes are set, the work on the edges starts. An iteration over all the edges is made, validating interval and type from filters and skipping if necessary. In case of working with paths, some special "restricted" edges filter applies. Restricted edges is a set that contains every edge with its direction. This set was built earlier when the edges were obtained from the database and is used to avoid inserting the same edge twice in different directions. Finally, the color is chosen, again taking into account if it is part of a path or not. If it is, it looks for the color of the nodes it connects and chooses the one with a higher color code, meaning that the last paths in the iteration have precedence over the first ones. If it is not part of a path, the specific color for that type of edge is used (which was selected in the settings view). Figure 19 shows the final result of the recreation of the visualization for the sample query in this project.

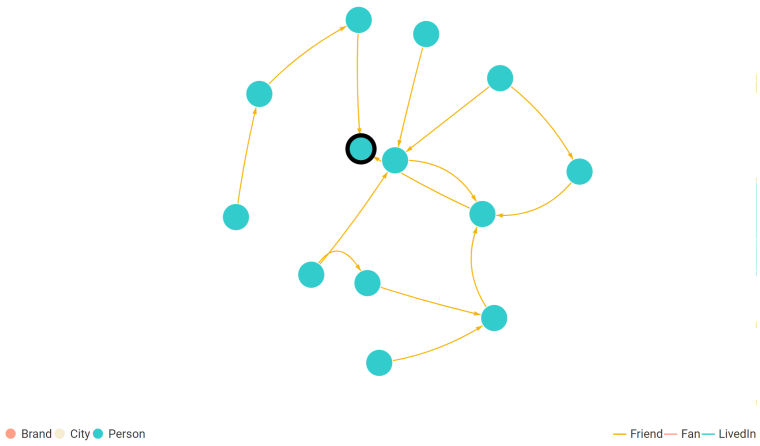

Figure 19: TGV sample visual representation

4.7 Color Criteria

To choose the color criteria when drawing the graphs, specifically when drawing paths, inspiration was taken from Cynthia Brewer. Among other things, an online tool for choosing color palettes was developed by her [21], as she believes that choosing effective color schemes to represent data is complex. There are three types of schemes, which are: sequential, diverging and qualitative. In this project a qualitative palette was selected because it works better to differentiate the different paths as the colors are more contrasting. To be more specific, the palette which was chosen is the 12-class paired, with 12 different colors. The

important thing to clarify is that when there are more than 12 different paths to show, the colors repeat, meaning that path 13 has the same color as path number 1.

Figure 20: 12-class paired color palette by Cynthia Brewer.

5 Implementation Details

This section explains some details on how the implementation deals with some problems, like granularity, integration and query processing.

5.1 Granularity

Some interesting fact about how the database actually works is that it can handle different temporal granularities at the same time, even in the same result. This means that different elements, nodes or edges, can have their time property set to anything in between years and seconds. This becomes clear when filtering the results. The year "2020" is greater than day "20-10-2019" but a comparison can not be made between "2020" and "20-20-2020". To do so, you must convert them into the same granularity.

So to correctly compare these two values, a conversion must take place. This is, in fact, easier when the context of this conversion is taken into account because the comparisons in between the values of the graph elements is not needed and the only focus is on the temporal filtering.

Time filtering is triggered through the slider having two extremes, one for the minimum and one for the maximum. So the only conversions are the minimum to the minimum that is allowed for that granularity completion, and the maximum to the max. This is done with a completion string of the complete granularity: *"-01-01 00:00:00"* and *"-12-31 23:59:59"* for max. Depending on the precision of the granularity more or less characters from the string are completed.

After this is done, two conversions into timestamp values occurs, in order to compare precisely with a numeric value. This is the clue to understand that the problem of conversion in time properties for the graph elements can be faced in the same way. Time properties are intervals so they have both a minimum and a maximum, which allows to use the same mechanism.

In the end, there are two intervals, which are completed to the same granularity and are converted into timestamps, leaving just a simple comparison to evaluate if the temporality of the element has some overlap with the temporality of the filter.

5.1.1 The value "NOW"

Most of these databases actually have moving time intervals and they address this characteristic with the value "now" for the upper bound of intervals. This is not compatible with the string completion solution but it is easily solved by

tweaking the interval comparison formula to interpret the value "now" as the greatest number allowed by the database system.

5.2 Integration

This subsection intends to explain the integration with the T-GQL engine developed in [1] and create a user-friendly interface.

5.2.1 Embedding

The first thing is to understand the way TGQ-L processor works in order to use it inside this solution. After a meeting with the developers and reviewing the code, interaction interface was chosen to be the integration point. Since the previous project has its way to input their queries, this project has to connect its the platform to that input.

To avoid making big changes, this project hooks up to their back end through an endpoint, using it as service provider. This is very handy since it avoids creating interfaces to interact between both projects, which generates constrains for languages, technology and more time spent on non-user features.

5.2.2 Compatibility & Reconstruction

The above solves the problem of integrating the two projects but it forces to handle the information which is returned from the T-GQL engine in a different way. The T-GQL engine works processing queries and returning, in JSON format, results to those particular cosults. On the other side our goal is to visually represent everything related with that result.

As expected, the query result is not enough to reconstruct the whole graph visually, since results are just nodes. So edges are missing. To solve this issue several queries need to be made directly to the database to support the reconstruction of the missing information related to the result. This was suggested by the colleagues from the other project. So a collection of all the nodes that are present in the result is created and is used to consult through a single Neo4j query for the corresponding edges. Although a bit more expensive this returns the information necessary to reconstruct most of the results. The reconstruction process is further explained on Section 4.6.

5.2.3 Paths

Taking into account the lack of information in the previous project's results, to construct path queries the order of the sequence in which the collection of nodes is presented is used to infer the directions of the paths.

Moreover, to reference different paths, different colors were used, but after several iterations, this feature was rolled back due to overlapping of nodes. Some results with several paths, have multiple paths that coincide in multiple nodes. So, having overlapping nodes, it is difficult to select the predominant color between this paths. This causes confusion. So it was decided to leave colors referencing only their corresponding node types and rely only in edges arrows to distinguish different paths.

5.3 Preprocessing the Query

Previous to the building of the graph, a preprocessing of the query needs to be done in order to obtain useful results to then generate the graph.

First, attributes are stripped out from the select clause, meaning that the following query:

```
SELECT c.Name
MATCH (c:City)
```

is transformed into:

```
SELECT c
MATCH (c:City)
```

The reason behind this is that with the first query, the TDBG returns the list of nodes, but the ids of those nodes corresponds to the id of Value nodes. To correctly obtain the involved edges in the query, the ids of the Object nodes are the ones needed. So by converting c.Name to c, the TDBG now returns the ids of the Object nodes and with that list of ids the edges can be retrieved as explained earlier. For this project, it does not matter which attribute the user wishes for, because the graph when displayed shows all the attributes in the selector module.

This stripping of the select clause is done by obtaining the index of the start of the select clause, the index of the start of the match clause and then getting the sub-string in between of these indexes. Then a split is done using the coma as the separator. Now every element corresponds to a an item of the select clause. Finally, every item is searched for a dot, and when found from the dot on wards everything is sliced. So every item now has only object and no attributes. The select clause is built again with the new items.

Second, a pre-processing of the select clause is made to find all the items wanted by the query to then highlight them when building the graph. To achieve this, the following two custom regex functions are used:

```
1- /\s?(\w+.?\w+|\w+\[\w+\])\s?=\s?('[\w\s-._]+'|\d+)/g
2- /(\w+).?\[?(\w+)\]?\s?=\s?('[\w\s-._]+'|\d+)/
```

The first one is used to obtain the where clause and the second one extracts every value wanted in the query. This values are stored in a array that is then used to highlight those wanted nodes.

6 Experimental Evaluation

To analyze the power of the application a thorough plan to test the performance under different scenarios was designed, which includes different databases with varying amounts of nodes and edges as well as a number of queries which were designed specifically to test each possible aspect. Each of these queries were then executed on the different databases and the execution times extracted, to make an analysis on the performance of each query. Two times were computed for every query. The first one starts when the processing of the query starts and ends when the graph is built. It includes the time it takes to parse the query, the time involved in asking the TDBG for a result of that query and finally

	Social Network 1	Social Network 2	Social Network 3
Nodes	392	1950	2100
Edges	1348	13537	23870
Parameters:			
Persons	70	500	500
Cities	30	50	100
Brands	30	100	100
Max friendships	5	25	100
Max friendship intervals	2	2	2
Max fans	2	25	10
Max fan intervals	2	2	2
Number of cPaths	2,1	2,1	2,1
cPath min Length	5,10	5,10	5,10

Table 2: Parameters used in the creation of each database.

the building of the graph. The second one is the the time involved in asking the TDBG for a result of that query. Then, these times are subtracted from to obtain the difference to compare how much processing this project adds.

6.1 Social network

Three different social network databases were created with an increasing number of nodes and edges. Below in Table 3 the different parameters of each database are shown:

In the following list, the five different queries used to analyze the social network database are listed, along with a code that is used when showing the response times for each of them:

```
SN0 -   SELECT p, c, n
        MATCH (p:Person),(c:City),(n:Brand)

SN1 -   SELECT paths
        MATCH (p1:Person), (p2:Person),
        paths = cPath((p1)-[:Friend*2..3]->(p2))
        WHERE p1.Name = 'Hilton Turner' and p2.Name = 'Jan Dickens'

SN2 -   SELECT c.Name , p1.Name, p2.Name
        MATCH (p1:Person)-[:Friend]->(p2:Person),
        (p2)-[:LivedIn]->(c:City)
        WHERE p1.Name = 'Hilton Turner'
        BETWEEN '2000' and '2004'


SN3 -   SELECT p2.Name as friend_name, p1
        MATCH (p1:Person)-[:Friend]->(p2:Person)
        WHERE p1.Name = 'Hilton Turner'
        WHEN
        MATCH (p1)-[e:LivedIn]->(c:City)
        WHERE c.Name = 'Danaeview'
```

```
SN4 -   SELECT paths
        MATCH (p1:Person), (p2:Person),
        paths = cPath((p1)-[:Friend*2]->(p2))
        WHERE p1[id] = 250
```

To get a better understanding of what each query does, an explanation for each of them is needed. SN0 just searches for all the people, cities and brand in the database. SN1 searches for all the friendships of distance 2 to 3 between Hilton Turner and Jan Dickens. SN2 searches for all the friends from Hilton Turner and the cities they lived between 2000 and 2004. SN3 searchers for all the friends from Hilton Turner when he was living in Danaeview. Finally, SN4 searches for all the friendships of distance 2 from the person with id 250.

Taking all this into account results are reported. Figures 21, 22, 23 show the response times of each query in the social network 1, 2 and 3 respectively. Each figure shows times for TGV (meaning the time it takes to do query plus build the graph) and for the TDBG (the time needed just to query the graph). The column nodes on the TGV table shows the amount of nodes that the query gives as result. The idea behind this is to compare both times and understand how much more processing the application adds.

Social Network 1										
	TVG (ms)						TDBG (ms)			
	Nodes	T1	T2	T3	AVG			T1	T2	T3	AVG
SN0	130	1597	1319	1323	1413		SN0	1213	1286	1306	1268
SN1	3	73	102	84	86		SN1	38	59	47	48
SN2	4	59	108	88	85		SN2	21	67	50	46
SN3	2	105	112	90	102		SN3	64	67	49	60
SN4	3	98	178	160	145		SN4	65	124	109	99

Figure 21: Response times for social network database 1.

Social Network 2										
	TVG (ms)						TDBG (ms)			
	Nodes	T1	T2	T3	AVG			T1	T2	T3	AVG
SN0	Not working						SN0	Not working			
SN1	5	119	216	185	173		SN1	83	170	139	131
SN2	14	272	217	152	214		SN2	199	157	105	154
SN3	5	338	157	105	200		SN3	290	109	59	153
SN4	109	2822	3073	2999	2965		SN4	2697	2965	2868	2843

Figure 22: Response times for social network database 2.

	TVG (ms)						TDBG (ms)			
	Nodes	T1	T2	T3	AVG			T1	T2	T3	AVG
SN0	Not working						SN0	Not working			
SN1	19	403	520	492	472		SN1	360	474	448	427
SN2	25	199	166	133	166		SN2	141	82	94	106
SN3	6	62	143	104	103		SN3	23	88	58	56
SN4	258	9596	12253	12158	11336		SN4	9257	11110	10986	10451

Figure 23: Response times for social network database 3.

The first thing to notice is that in social network 2 and 3, SN0 does not work. This happens because of the increase in the maxFriendship parameter. In social network 1, maxFriendship is 5 while in 2 and 3, maxFrindship is equal to 25 and 100 respectively. 5 and 20 times more.

Second, the response times in SN0 and SN4 are much greater than in the other queries, surpassing the 1000 milliseconds. The reason behind this is the

amount of nodes the response has. It can be noted that over the 100 nodes, the response time is greater than 1 second.

Going into detail regarding the difference in time between TVG and TDVG, Figure 24, shows specifically this.

Difference between TVG and TDBG (ms)			
	Social Network 1	Social Network 2	Social Network 3
SN0	145		
SN1	38	43	44
SN2	39	60	60
SN3	42	47	47
SN4	46	121	885

Figure 24: Difference between the average TVG and TDBG response times.

For SN1 to SN4, the difference in response times is nearly constant around 40 milliseconds in Social Network 1. All these queries have less than 10 nodes as result. The same happens for SN1 and SN2 in Social Network 2 and 3. For SN2 in Social Network 2 and 3, the difference is around 60 milliseconds, and the result nodes are greater than 10. Finally, for SN0 in Social Network 1 and SN4 in Social Network 2 and 3, the difference is greater than 100 as the result nodes are greater than 100 too.

Form these results, a conclusion starts to be noticeable. This is that as the amount of nodes in the result of the query increases, so does the difference between the response times in TVG and TDBG. For similar numbers of nodes, the difference is constant, which shows the consistency of this project when building the graph. Moreover, In every case, the difference is lower than 1 second.

For a better understanding of the above, Figures 25, 26 and 27 were made about the results obtained.

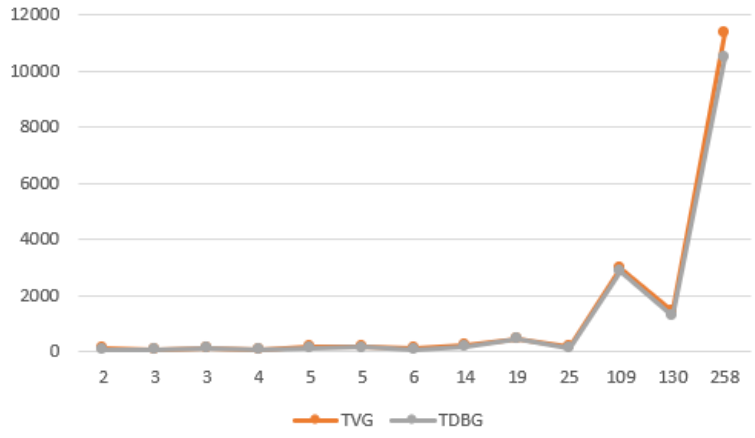

Figure 25: Response times depending on the number of nodes.

Figure 25 shows both TVG and TDBG response times depending on the number of nodes. All the queries for the 3 databases are present in this graph.

Both lines in this graph are overlapping each other, meaning that the difference in response times is small.

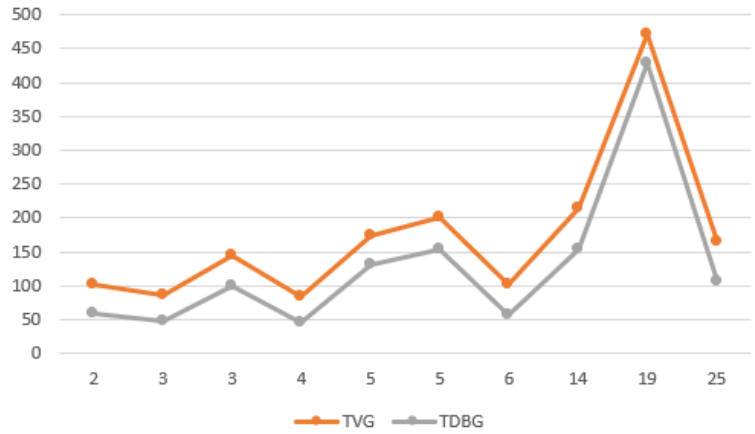

Figure 26: Response times depending on the number of nodes up to 100 nodes.

Figure 26 shows the same graph as Figure 25 but for the first results with less than 100 nodes. This graphs shows that the difference between the lines is almost constant all the time showing the consistency previously mentioned.

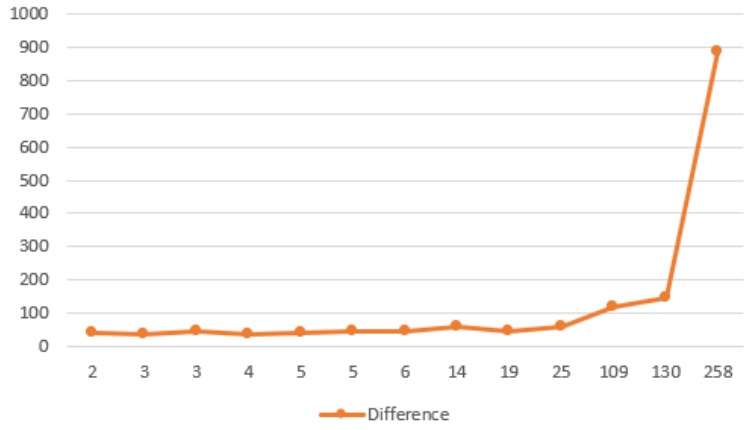

Figure 27: Difference between the average TVG and TDBG response times depending on the number of nodes.

Finally, Figure 27 shows the difference between the average TVG and TDBG response times depending on the number of nodes. All the queries for the 3 databases are present in this graph. It is clear that below the 100 nodes, the line is almost straight, again showing the consistency intended. And the max difference in the graph is less than 1000 milliseconds which is less than a second and almost imperceptible for humans.

	Airport 1	Airport 2	Airport 3
Nodes	60	600	1800
Edges	82	788	2400
Parameters:			
Cities	10	100	300
Outgoing flights per airport	3	6	9
Flights per destination	3	6	9

Table 3: Parameters used in the creation of each database.

6.2 Airport

For further testing, three different airport databases were created with an increasing number of nodes and edges. Below the different parameters of each database are shown:

In the following list, the 5 different queries used for airport databases are enumerated with a code that is used when showing the response times for each of them:

```
A0 -    SELECT a, c
        MATCH (a:Airport), (c:City)

A1 -    SELECT path
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport),
        (c2:City)<-[:LocatedAt]-(a2:Airport),
        path = fastestPath((a1)-[:Flight*]->(a2))
        WHERE c1.Name = 'Port Berniecebury'
        AND c2.Name = 'West Hipolitohaven'

A2 -    SELECT path
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport),
        (c2:City)<-[:LocatedAt]-(a2:Airport),
        path = latestDeparturePath((a1)-[:Flight*]->(a2),
        '2020-12-11 17:04')
        WHERE c1.Name = 'Port Berniecebury'
        AND c2.Name='West Hipolitohaven'

A3 -    SELECT a1, a2, c1
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport)-[:Flight]
        ->(a2:Airport)
        WHERE c1.Name = 'Port Berniecebury'

A4 -    SELECT a1, a2, c1
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport)-[:Flight*4]
        ->(a2:Airport)
        WHERE c1.Name = 'Port Berniecebury'
```

To get a better understanding of what each query does, we will explain them one by one. A0 is the first query and just searches for all the airports and cities in the database. A1 searches for the fastest paths between the airports

located in the cities of Port Berniecebury and West Hipolitohaven. A2 searches for the lastest departure path since 17:04 in 2020-12-11 between the previously mentioned cities. A3 searchers for all the airports and cities which have direct flights from Port Berniecebury. Finally, A4 searches for all the airports and cities at a 4-scale distance from Port Berniecebury.

Taking all this into account results are now exposed. Figures 28, 29, 30 show the response times of each query in the airport 1, 2 and 3 respectively. Each figure shows times for TGV (meaning the time it takes to do query plus show build the graph) and for the TDBG (the time needed just to query the graph). The column nodes on the TGV table shows the amount of nodes that the query gives as result.

Airport 1

	TVG (ms)						TDBG (ms)				
	Nodes	T1	T2	T3	AVG			T1	T2	T3	AVG
	20	78	123	94	98		A0	56	98	76	77
A1	3	41	87	106	78		A1	27	62	91	60
A2	4	182	102	137	140		A2	147	83	126	119
A3	3	111	74	84	90		A3	79	63	68	70
A4	11	158	123	135	139		A4	124	107	121	117

Figure 28: Response times for Airport database 1.

Airport 2

	TVG (ms)						TDBG (ms)				
	Nodes	T1	T2	T3	AVG			T1	T2	T3	AVG
A0	200	508	408	476	464		A0	442	362	391	398
A1	2	93	59	77	76		A1	51	44	60	52
A2	2	134	106	118	119		A2	122	89	106	106
A3	8	93	61	73	76		A3	68	51	61	60
A4	45	189	132	129	150		A4	157	122	118	132

Figure 29: Response times for Airport database 2.

Airport 3

	TVG (ms)						TDBG (ms)				
	Nodes	T1	T2	T3	AVG			T1	T2	T3	AVG
A0	600	2881	2819	2829	2843		A0	2605	2762	2773	2713
A1	2	38	76	71	62		A1	26	57	51	45
A2	2	131	85	87	101		A2	121	63	76	87
A3	6	35	84	75	65		A3	24	63	65	51
A4	36	182	259	289	243		A4	169	245	275	230

Figure 30: Response times for Airport database 3.

The first thing to noticed is that unlike in social network databases, A0 works. This is because the parameters outgoingFlights and flightsPerDestination are not that big. This was done on purpose to analyse the different response times in the first query, that searches for everything, as in social network it was not been possible to test.

Secondly, these databases show much better response times than the social network databases. All the response times are below 1000 milliseconds except for A0 in Airport 3. Even in the case of A0 in Airport 2 where the mount of nodes is 200. This improvement is because of the maxFlights parameter. As each node has clearly less relationships than in social network, queries are much faster.

Going into detail regarding the difference in time between TVG and TDVG, Figure 31 shows specifically this.

Difference between TVG and TDBG (ms)			
	Airport 1	Airport 2	Airport 3
A0	22	66	130
A1	18	25	17
A2	22	14	14
A3	20	16	14
A4	21	18	14

Figure 31: Difference between the average TVG and TDBG response times.

Again, a clear pattern can be found, where queries with less than 10 nodes have response times around the 20 milliseconds. Although in these databases, even with 36 nodes response times remain in the 20 millisecond mark. which is an improvement over the other case. For 200 nodes, the difference increases to 66 and for 600, to 130. Conclusions from this test are the same as with social network databases. As the number of nodes increases, the difference in response times does too, though, the increase is less steep. For similar numbers of nodes the difference remains constant. Further, in this case, the maximum difference between response times is 130 milliseconds for 600 nodes.

The case of A0 in Airport 3 is the reason why the parameter maxFlights is less than 10 in every database. To test the performance with big amount of nodes. And the results are better than expected. Less 150 millseconds difference between the two projects for 600 nodes and a total time of nearly 3 seconds to show 600 nodes.

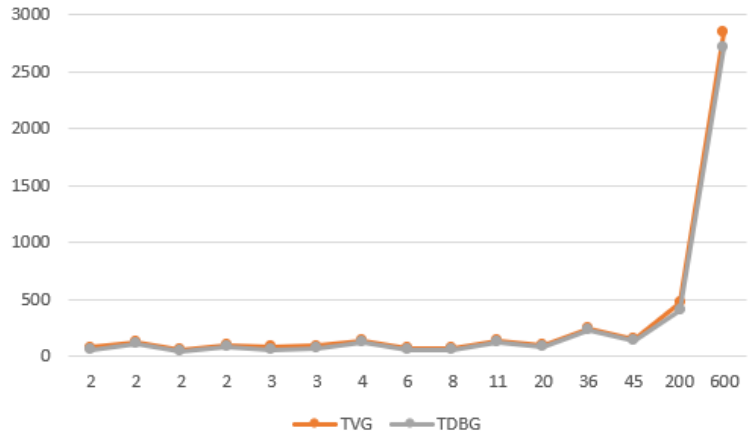

Figure 32: Response times depending on the number of nodes.

Figure 32 shows both TVG and TDBG response times depending on the number of nodes. All the queries for the 3 databases are present in this graph. Both lines in this graph are overlapping each other, meaning that the difference in response times is small.

Figure 26 shows the same graph as Figure 33 but for the first results with less than 100 nodes. This graphs shows that the difference between the lines is

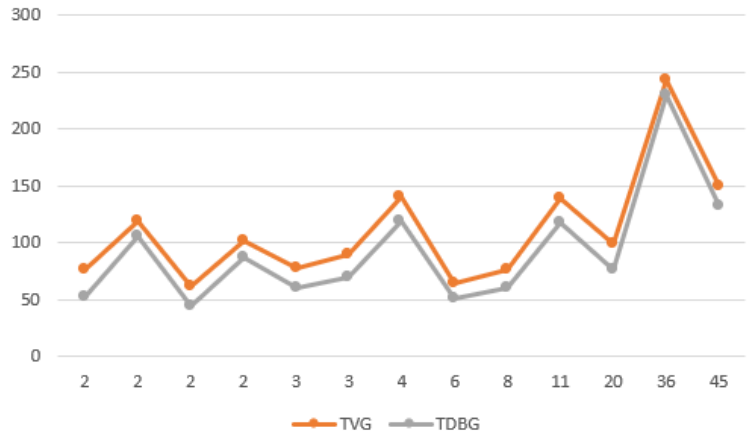

Figure 33: Response times depending on the number of nodes up to 100 nodes.

almost constant all the time showing the consistency previously mentioned.

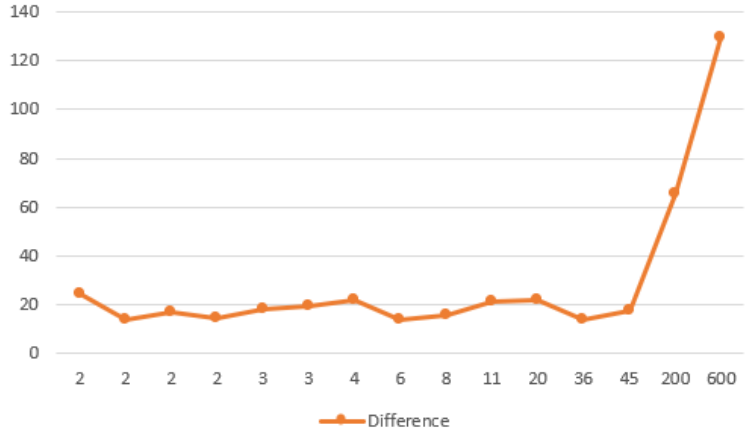

Figure 34: Difference between the average TVG and TDBG response times depending on the number of nodes.

Finally, Figure 34 shows the difference between the average TVG and TDBG response times depending on the number of nodes. All the queries for the 3 databases are present in this graph. Unlike in social network databases, below the 100 nodes, the line is less straight. Still the difference between values is low, less than 10 milliseconds. The max difference in the graph is less than 140 milliseconds as said before.

7 Limitations

7.1 Working With Deeply Interconnected Networks

This might not be related to this project directly but as the querying engine from T-GQL is fully used, results are limited to queries that work in there. Any limitation regarding the T-GQL project is extent towards this project, since no further work was provided on the T-GQL project.

7.2 Different Path Coloring

As it was mentioned earlier, the decision to roll back path coloring to distinguish different paths when shown in the same query result had to be made. This is a clear limitation to show the ideal solution where everyone can easily distinguish between them. A feasible way to do this could not be found but there can be some other solution to implement if some clicking behaviour and highlight in the path is added. Of course this is not an easy thing to do, due to the overlapping edges, which would generate ambiguous selections in some cases.

8 Conclusion and Open Problems

The platform and visualization work described throughout this document proved to become a useful tool for exploring and analyzing temporal property graphs. Regarding the visualization solution proposed, the main important points were solved. The abstraction of the underlying structure of the temporal graphs was successfully performed through the model proposed. Integration with the T-GQL engine was solved to full extent, empowering users with the possibility of using T-GQL language and hooking visualizations to the results. The platform is also considered a success since users can easily work through it and use the tools with ease. Several usability concerns where tackled, not only through the right color criteria for visualization, but also by paying close attention to components such as the starting query generator with fuzzy search.

8.1 Color Scales for Temporality Density

One feature to include is a density scale to gain more insight from result temporal distribution. This can provide further information to the user without using any further screen space. The idea behind this feature is to count the amount of nodes and/or edges for every time block in the whole time interval. Once the "histogram-like" information is obtained, the next step is to flatten this down towards a color scale to show this on the same line of the slider.

After working on the implementation strategies available for the feature, it is clear that all the data cooking process can become quite costly, since the information for temporality is returned as string types referencing dates, so dumping this into a scale is not trivial. Nevertheless, this feature can be implemented in future developments, since there is probably a moment during graph reconstruction where temporality values are converted into timestamps, so this information can be leveraged to get the distribution information.

Appendices

A Visualization Research Extension

A.1 OGMA

This first one that has been mentioned is a Javascript library designed for large-sacle interactive graph visualizations[22]. It is designed with WebGL but it does support HTML5 Canvas and SVG. Its features go from displaying to interactin with graph data, embedded in a web application. The issue with this project is that although all the information is public and seemed to be accessible for our work, this is a commercial library and they are not giving any academic license at the moment.

A.2 G6

Coming to G6 [23], this D3 superlibrary is very well documented and is actually open-source. This typescript library has been designed as a graph visualization engine, which provides a set of basic mechanisms, including rendering, analysis, interaction, and other auxiliary tools.

Although documentation is detailed, everything is clearly written, first, in Chinese and then translated. This generates some comprehension complexities that make implementation for this tool, quite a struggle. Furthermore, nodes data handling by G6 is very visually precise, so every position has to be set by indicating their coordinates. Viewing it from a visualization perspective this is a really good feature but handling space distribution adds another complexity for our platform engine, which is not under the scope for this project.

A.3 VX

The last one is a React wrapper library for a subset of D3 components[24]. This was by far the most flexible library that was actually easy to support inside our React platform. As D3 this library has no complete solution for graph visualization but it has several small components that give the granularity needed to implement our solution, focusing on logic and not on the actual graphic implementation. Nevertheless making all this smaller components to work and reach a useful visualization requires lots of time spent on understanding the library and every component functions. Since our main goal is to solve visualization problems from temporal property graphs, most of the time needs to be saved for it.

B React Template

A React template from Creative Tim [14] was chosen as a starting point. In past experiences we've come into his contributions and we've found that his work had a very clear structure and good documentation for further development.

[14] Any of his templates, either free and paid, can be found at https://www.creative-tim.com/

Since this was just the base of our platform, clear documentation was really important and a key to build up the rest of the platform.

References

[1] Ariel Debrouvier, Eliseo Parodi Almaraz, and Matías Perazzo. 2020. *Temporal information query language for graph databases*. Master's thesis. CABA, Argentina.

[2] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman, editors. 1999. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN: 1558605339.

[3] Manuel Lima. 2017. *The Book of Circles: Visualizing Spheres of Knowledge*. Princeton Architectural Press, (May 2017).

[4] Moshe Bar and Maital Neta. 2006. Humans prefer curved visual objects. *Psychological Science*, 17, 8, (August 2006), 645–648. DOI: 10.1111/j.1467-9280.2006.01759.x.

[5] O. Vartanian, Gorka Navarrete, A. Chatterjee, L. B. Fich, H. Leder, C. Modroño, M. Nadal, Nicolai Rostrup, and M. Skov. 2013. Impact of contour on aesthetic judgments and approach-avoidance decisions in architecture. *Proceedings of the National Academy of Sciences*, 110, (April 2013), 10446–10453. DOI: 10.1073/pnas.1301227110.

[6] John N. Bassili. 1978. Facial motion in the perception of faces and of emotional expression. *Journal of Experimental Psychology: Human Perception and Performance*, 4, 3, 373–379. DOI: 10.1037/0096-1523.4.3.373.

[7] Manuel Lima. 2009. Information visualization manifesto. (August 2009). http://www.visualcomplexity.com/vc/blog/?p=644.

[8] Sala. 2006. Websites as graphs. http://www.aharef.info/2006/05/websites_as_graphs.htm.

[9] Mike Bostock. 2017. Force-directed graphs. (November 2017). https://observablehq.com/@d3/force-directed-graph?collection=@d3/d3-force.

[10] Neo4j-contrib. 2016-2020. Neovis.js. https://github.com/neo4j-contrib/neovis.js/.

[11] Cambridge Intelligence. 2021. The regraph toolkit. https://cambridge-intelligence.com/regraph/.

[12] K. Semertzidis and E. Pitoura. 2019. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31, 1, 181–194. DOI: 10.1109/TKDE.2018.2823754.

[13] Wenyu Huo and Vassilis J. Tsotras. 2014. Efficient temporal shortest path queries on evolving social graphs. In *Conference on Scientific and Statistical Database Management, SSDBM, Aalborg, Denmark, June 30 - July 02, 2014* Article 38. ACM, 1–4. DOI: 10.1145/2618243.2618282.

[14] Anja Naumann, Jörn Hurtienne, Johann Israel, Carsten Mohs, Martin Kindsmüller, Herbert Meyer, and Steffi Husslein. 2007. Intuitive use of user interfaces: defining a vague concept. In (January 2007), 128–136. ISBN: 978-3-540-73330-0. DOI: 10.1007/978-3-540-73331-7_14.

[15] Charlie Kreitzberg. 2017. The intuitive interface. (February 2017). https://ux.princeton.edu/learn-ux/blog/intuitive-interface.

[16] Renzo Angles. 2018. The property graph database model. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018* (CEUR Workshop Proceedings). Volume 2100. CEUR-WS.org.

[17] Texcel Research Jonathan Robie. 1998. Rec-dom-level-1. (October 1998). https://www.w3.org/TR/REC-DOM-Level-1/introduction.html.

[18] Vis.JS. 2018. Vis-network. https://visjs.github.io/vis-network/docs/network/index.html.

[19] Prakash Patel and Samvid Mistry. 2016. A guide to material design, a modern software design language, (April 2016), 65–66.

[20] Jacques Bertin. 1983. *Semiology of Graphics*. University of Wisconsin Press. ISBN: 0299090604.

[21] Cynthia Brewer. 2002. Colorbrewer 2.0. Colorbrewer2.org.

[22] Linkurious. 2013. Ogma. https://doc.linkurio.us/ogma/latest/.

[23] AntV Team. 2018. G6: a graph visualization framework in typescript. https://github.com/antvis/G6.

[24] hsoff. 2017. Vx = react + d3. https://vx-demo.now.sh/gallery.