TGV: A Visualization Tool for Temporal Property Graph Databases

Peer-reviewed author version

Orlando, Diego; Ormachea, Joaquin; SOLIANI, Valeria & Vaisman, Alejandro Ariel
(2024) TGV: A Visualization Tool for Temporal Property Graph Databases. In: Information systems frontiers, 26 (4), p. 1543-1564.

DOI: 10.1007/s10796-023-10426-1
Handle: http://hdl.handle.net/1942/41529

TGV: A Visualization Tool for Temporal Property Graph Databases

Preprint · August 2023

CITATIONS
0

READS
75

4 authors, including:

Valeria Soliani
Instituto Tecnológico de Buenos Aires
7 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Alejandro Vaisman
Instituto Tecnológico de Buenos Aires
154 PUBLICATIONS **2,962** CITATIONS
SEE PROFILE

Noname manuscript No.
(will be inserted by the editor)

TGV: A Visualization Tool for Temporal Property Graph Databases

Diego Orlando · Joaquín Ormachea · Valeria Soliani · Alejandro Vaisman

Received: date / Accepted: date

Abstract Graph databases are increasingly being used in the data science field, in particular to represent different kinds of networks. In real-world situations, the nodes and edges in a network evolve across time. For example, in a social network, people's preferences and relationships change, as well as the characteristics of the network entities themselves. Temporal property graph databases aim at capturing these changes, by means of appropriate data models and query languages that allow users to represent, store, and query time-varying graphs. In order to exploit their full potential, temporal property graph databases require visualization tools that allow navigating graph data across time. To address this need, the present work introduces a framework for temporal property graph visualization, denoted TGV, based on T-GQL, a data model and query language for temporal graphs implemented over a Neo4j, a widely-used graph database. TGV allows editing and running T-GQL queries, displaying the result, and navigating such result across time. Further, TGV displays temporal graphs in a transparent way, hiding the underlying T-GQL structure from the user.

Keywords Graph Visualization · Temporal Graphs · Temporal Database · Neo4j

1 Introduction

Property Graphs [2, 11, 20] have been increasingly gaining popularity, especially for modeling and analyzing different kinds of networks. In short, prop-

Diego Orlando, Joaquín Ormachea, Alejandro Vaisman
Department of Information Engineering, Instituto Tecnológico de Buenos Aires Lavardén 315, C1437FBG, Ciudad Autónoma de Buenos Aires, Argentina
Tel.: +5411-3754-4864
E-mail: {dorlando,formachea,avaisman}@itba.edu.ar
Valeria Soliani
Department of Information Engineering, Instituto Tecnológico de Buenos Aires and Hasselt University, Belgium E-mail: vsoliani@itba.edu.ar

erty graphs are graphs whose nodes and edges are annotated with attributes, denoted properties. The property graph data model underlies most graph databases in the marketplace [1]. Examples of graph databases based on this model are Neo4j[1] and Janusgraph[2]. Although time is present in most real-world applications, most efforts in the field consider graphs as static, that is, as of a certain moment in time (usually, the current time). Many different kinds of changes may occur in a property graph as the world they represent evolves over time: edges, nodes and properties can be added and/or deleted, and property values can be updated, to mention a few. For instance, in a phone call network, where each vertex represents a person (or a phone number) and edges represent calls between them, new nodes and edges are added frequently and also the properties of the nodes may change over time. As another example, in social networks, where each vertex models a person or organization, and an edge represents a relationship between two such entities during a time interval, relationships and entities may also change at any time. Ignoring the time dimension could lead to incorrect results or prevent interesting analysis possibilities. For example, it may be relevant to know the interval of the relationships that occur in a social network, to weight their strength, or to find out chains of relationships that occurred simultaneously. For example, a user may be interested in asking for "Favourite foods of Mary while she was living in Argentina." Those are queries that could not be answered without accounting for time. As another example, in a transportation network, a planning engineer may ask for the "Time saved for going from Buenos Aires to Córdoba after the construction of Highway Number 11."

In light of the above, some works have proposed to extend property graphs with the capability of keeping and querying their history. Debrouvier et al. [10] propose a model and a query language (T-GQL) aimed at representing and querying the evolution of a (property) graph across time. The proposal applies temporal databases concepts [22] to graph databases, in order to model, store, and query temporal property graphs. In the data model, nodes, relationships, and node properties are timestamped with their temporal validity interval, and graphs are heterogeneous, meaning that relationships may be of different kinds. These graphs are called Interval-labeled Property Graphs. Temporal graphs not only address the representation of the history of a graph, but also allows defining several different path semantics. In [10], three such semantics are considered (besides the traditional one): continuous paths, pairwise continuous, and consecutive path semantics. The first one captures paths that are valid continuously during a time interval (e.g., if Mary was a friend of Peter and Peter a friend of Peggy, between 2019 and 2021, then there is a continuous path [Mary, Peter, Peggy] with interval [2019,2021]). The second one relaxes this constraint, allowing a pairwise overlap between intervals ((e.g., if Mary was a friend of Peter between 2017 and 2019, and Peter a friend of Peggy between 2018 and 2021, then there is a pairwise continuous path [Mary, Peter, Peggy]). Finally, if there is a path with no overlap, but where time intervals are consecutive, this path is denoted consecutive (e.g., if there is a flight from Paris to Rome, and one hour later the arrival to Rome, a flight from Rome to

[1] `http://www.neo4j.com`

[2] `http://janusgraph.org/`

Athens, then there is a consecutive path [Paris, Rome, Athens]). Consecutive paths can be of different kinds: shortest, earliest-arrival paths, latest-departure paths, among other ones [8]. T-GQL is a high-level query language for graphs, that allows expressing queries like the ones mentioned above. The authors also present a Neo4j-based implementation of T-GQL, and a client interface allowing to write T-GQL queries. In this implementation, the underlying graph is a Neo4j graph, where nodes and edges encode temporal information. This underlying structure remains hidden in T-GQL.

A relevant problem that arises when working with temporal graphs is how to display them in a way that can be useful and friendly to the users, regardless how data are actually stored. This is not a trivial problem, given the large amounts of information and the level of abstraction present in temporal graph databases. The present paper addresses this problem, building from the concepts and implementation developed in [10], where T-GQL queries are executed through a client tool called TDBG. The paper describes the design and implementation of a framework, denoted *Temporal Graph Visualizer* (TGV), for interacting with, and visualizing temporal graphs. The problem of visualizing temporal property graphs is approached in two ways. The first one focuses on the platform development and its functionality and the second one focuses on the visualization capabilities needed to correctly handle the interaction with the database. The work aims at providing a platform that encapsulates every functionality, making the underlying data structure transparent to the user. Developing a friendly interface for user interaction is another goal of this work. According to [15], a well-designed software should reduce the user's effort to think about the way of using such software. Thus, the idea of this work is to maintain a low cognitive load for the platform (allowing users to focus on the data they are analyzing), and also to reduce the training load needed to use the application. The platform presented here includes a panel allowing users to write their own queries in the T-GQL query language. This work provides, then, a visualization platform that integrates with the query engine described in [10].

1.1 Contributions

This paper describes the design and implementation of a web platform and tools for temporal property graphs visualization. Visualization capabilities come in two flavours: on the one hand, the user can write T-GQL query language in an application panel and the result is displayed graphically, in a way that she can navigate the graph across time, starting from the query result. For example, if T-GQL asks for the continuous paths in the temporal graph starting from a certain node, then only the related nodes will be displayed, and the user can start temporal navigation back and forth from this portion of the graph. On the other hand, starting from the whole graph, a slider bar allows navigating across time (certain tools that prune the initial graph are also provided). This paper (a) presents implementation details, emphasizing on the abstraction mechanism that hides the structural details of the temporal database and offers the user a clean view of the temporal graph; (b) gives ex-

amples of the use of the interface, either starting from T-GQL queries or from the whole graph; and (c) discusses two case studies that show how the visualizer works over two different kinds of problems, one referring to the history of a social network, and the other one to flight schedules. The experiments show that the overhead introduced by the visualization tool is not relevant, and most of the times negligible.

It is worth noting that, although there are applications that allow displaying graphs across time (these are mentioned in Section 2), they do not address the problem of dynamically querying a graph database and navigating the result (which is in turn a temporal graph) across time.

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 briefly describes the T-GQL language. Section 4 provides a general view and rationale of the visualizer, and explains how the temporal graph is built after a T-GQL query is submitted and the result returned. Section 5 shows how navigation across time is performed, and Section 6 reports the experiments carried out over the prototype, reporting their results. Section 7 concludes the paper.

2 Related Work

This section reviews existing work about general concepts in the fields of graph and information visualization.

2.1 Information Visualization

Visualizing information in a way that it is both informative and appealing has been a research topic for many years. Card et al. [9] define information visualization as the use of computer-supported, interactive, visual representations of abstract data to amplify cognition. According to Lima [17], there are three main key issues in Information Visualization.

1. Humans prefer curves: Humans show, since infancy, a preference from curves, a statement corroborated by Bar and Neta [3], which reveals a strong human preference for curved objects and typefaces. Also, in Vartanian et al. [23], a similar inclination in architectural spaces was reported.
2. Circles equal happiness: This theory is explained by an experiment by Bassili in 1978 [4], 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. The conclusion of this experiment is that expressions of anger show acute downward "V" shapes (angled eyebrows, cheeks, and chin), while expressions of happiness are conveyed by expansive, outward curved patterns (arched cheeks, eyes, and mouth).

3. Spherical geometry of the eye: This third hypothesis reveals how the circular framing and spherical geometry of a person's 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.

The so-called "Information Visualization Manifesto" [16] claims that any information visualization project should follow 10 rules. From these ten, two are the most relevant to the present work: (1) "Interactivity is key"; and (2) "Embrace time". The first explains that through interactive techniques users are able to properly investigate and reshape the layout in order to find appropriate answers to their questions. The second rule highlights that people can quickly realize that a snapshot in time only tells a small portion of information about the community. On the other hand, if time had been properly measured and mapped, it would allow a rich understanding of the changing dynamics of a given social group.

A very well-known visualization tool, *Observable*[3] is worth mentioning here. It has been created by Mike Bostock, who also developed the widely used Javascript library *d3.js*. Observable is a Javascript sandbox (which uses d3.js as the visualization tool) for code sharing and open collaboration in which non-specialized users can visualize data in real time.

2.2 Graph Visualization

Graph visualization is a particular branch of information visualization. The *Visual Complexity* platform,[4] is a unified resource space for the visualization of complex networks. Launched in 2005, its main goal is to leverage a critical understanding of different visualization methods, across a series of disciplines, as diverse as Biology, Social Networks and the World Wide Web. Further, Bostock publishes a notebook (called Temporal Force-Directed Graphs) [6] which allows visualizing a temporal network that changes over time, using Observable. As an example, in the top-left corner of the graphic on the upper part of Figure 1, a slider allows displaying the progression of the graph, with a 'play' button and the date from which data are obtained. A similar idea is used in the project presented in this paper.

NeoVis[5] is an open-source graph visualization tool created by eleven developers from Neo4j. This tool is built using vis.js, a Javascript library for Neo4j.[6] Given a query written in Cypher (Neo4j's high-level query language), a graph can be shown directly on any canvas. It also allows the developer to modify and combine features, like colours or families, to generate different groups. The lower part of Figure 1 shows a social network visualization using Neovis.

One of the most recent developments for graph visualization from Cambridge Intelligence, ReGraph [13], is a full plain graph visualization tool de-

[3] `https://observablehq.com/`

[4] `http://www.visualcomplexity.com/vc/`

[5] `https://github.com/neo4j-contrib/neovis.js/`

[6] `https://visjs.org/`

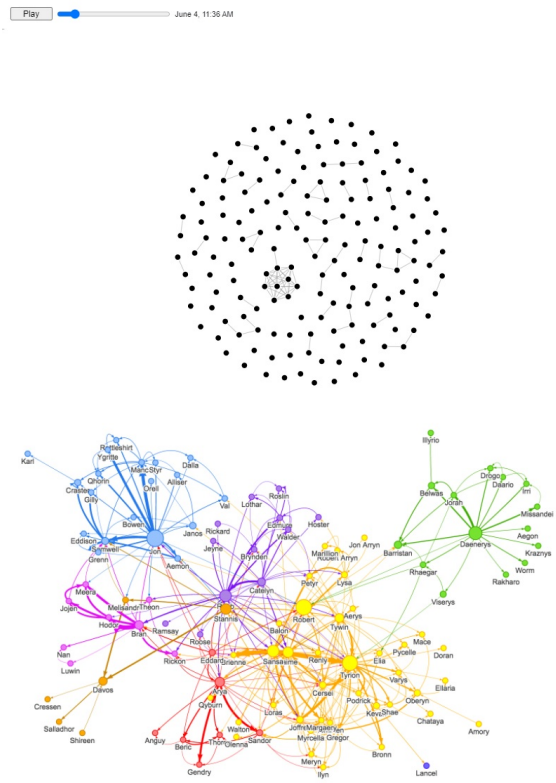

Fig. 1 Left: Temporal Force-Directed graph by Mike Bostock; Right: NeoVis visualization of a social network.

signed for React[7]. React is a popular framework for web development, and it is also used in the work described in the present paper. 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 is in the front end of the application. Data are passed on as a plain JavaScript object.

A plugin for creating a visualization network, denoted *d3-network-time*, is also available[8]. It can be used to animate the evolution of the network over time, or to display the network as of a specific point in time.

[7] `https://reactjs.org/`

[8] `https://github.com/dianaow/d3-network-time`

2.3 Temporal Graph Models

Temporal graphs represent the history of a graph. They can be classified as duration-labeled, interval-labeled and snapshot-based [10]. In the first class, edges are labeled with a value representing the duration of the relationship between the two nodes that the edge relates. The main use of this kind of temporal graphs is for scheduling problems, where some sort of shortest path must be computed, implementing some ad-hoc variation of the Dijkstra's algorithm. An interval-labeled temporal graph is a graph where each edge represents a relationship from a vertex to another one, valid during a time interval denoted as [ti,tf]. Valid time is considered in the remainder, that is, the time 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. In Snapshot-based temporal graphs [21,12], the history of a graph is given in the form of graph snapshots corresponding to the state of the graph at different time instants. Given a query, a relevant problem consists in efficiently finding those matches in the graph history that persist over time, that means, those matches that existed for the longest time, either contiguously (in consecutive graph snapshots) or not. 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.

The present paper is based on the work by Debrouvier et al. [10], where the authors define a temporal graph data model where nodes and relationships contain attributes (properties) timestamped with a validity interval. Graphs in this model can be heterogeneous, that is, relationships may be of different kinds. Associated with the model, the authors present a high-level graph query language, denoted T-GQL, together with a collection of algorithms for computing different kinds of temporal paths in a graph, capturing the temporal path semantics mentioned in Section 1, along with a Neo4j-based implementation. This model is explained in the next section.

3 The T-GQL Model and Language

In order to make the paper self-contained, this section presents in a streamlined fashion, the main notions of T-GQL data model and query language.

3.1 T-GQL Data Model

The temporal model used in this paper is composed of three kinds of nodes: *Object*, *Attribute*, and *Value* nodes. Every Object and Attribute node, and every edge in the graph are associated with a tuple (`name`,`interval`). The `name` represents the content of the node (or the name of the relationship), and the `interval` represents the period(s) during which the node is (was) valid, and it is a temporal element (i.e., a set of intervals). Analogously, Value nodes are associated with a (`name`,`interval`) pair. Value nodes contain the actual value of an attribute or property, in a certain interval.

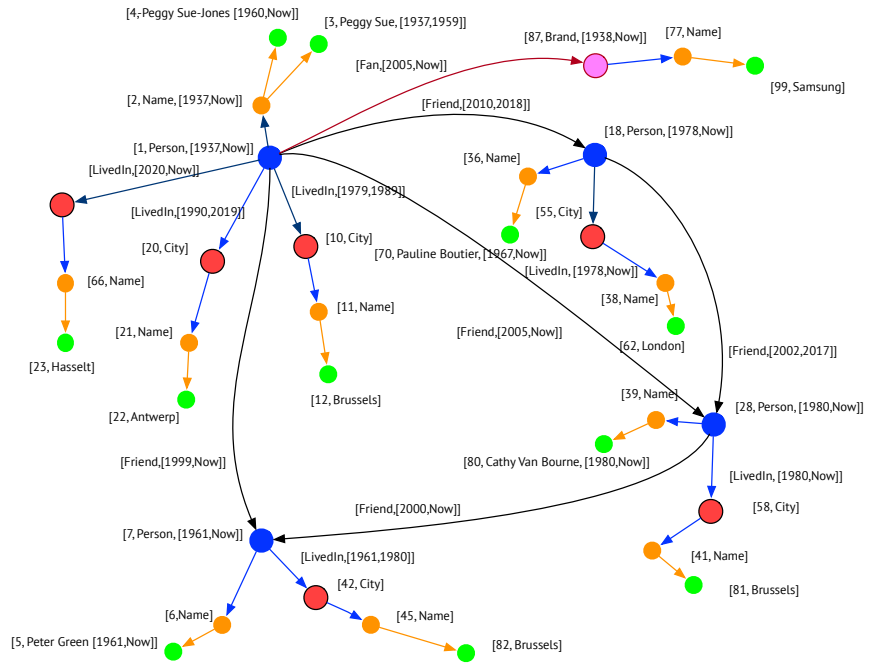

Fig. 2 Friend-of-a-Friend Temporal Graph

Figure 2 shows a social network represented in the temporal model introduced above. There are three kinds of Object nodes, namely Person, City, and Brand. There are also three temporal relationships: LivedIn, Friend, and Fan. The first one is labeled with the periods when someone lived somewhere. The second one is labeled with the periods when two people were friends. The third one is labeled with the periods when a person was fan of a certain brand. The Attribute node Name associated with a Person node represents the name of the person, and it is also temporal. The actual value of the Attribute node is represented as a Value node, e.g., the node in green with id=3 and value "Peggy Sue". Note that this value changes to "Peggy Sue-Jones", showing the temporality of the Attribute node Name. There are other Attribute nodes denoted Name associated with the other Object nodes in the graph. For clarity, if a node is valid throughout the complete history, the temporal labels are omitted.

A key issue when querying graphs is reachability, that is, the problem of computing all nodes reachable from a given one. Other well-known problems involve computing the shortest path between two nodes. In a temporal context, these problems become more involved, since different semantics can be used to compute reachability. The T-GQL model supports different kinds of temporal paths semantics, as mentioned in Section 1. The two ones shown in this paper are *continuous path* and *consecutive path* semantics. The former are paths that are valid continuously during a certain interval (as introduced in [19]). An example is given in Figure 3, where two continuous paths can be observed,

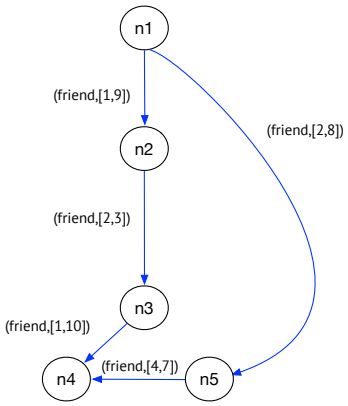

Fig. 3 Continuous paths [cf. [10]].

$(n_1, n_2, n_3, n_4, friend, [2, 3])$ and $(n_1, n_5, n_4, friend, [4, 7])$. That is, n_4 can be reached from n_1, traversing the edges labeled friend, during the interval $[2, 3]$ with a path of length 3, and during the interval $[4, 7]$ with a path of length 2. The interval when n_4 is continuously reachable from n_1, is obtained by taking the union of both intervals, that is $[2, 7]$.

Consecutive paths are paths consecutive in time whose intervals do not overlap. Consecutive path semantics is useful for defining time schedules. With the notion of consecutive path, several different temporal paths can be defined as follows [8]. The *earliest-arrival path (EAP)* is the path that can be completed in a given interval such that the ending time of the path is minimum. The *latest-departure path (LDP)* is the path that can be completed in a given interval such that the starting time of the path is maximum. The fastest (FP) is the path that can be completed in a given interval such that its duration is minimum. The *shortest path (SP)* is the path that can be completed in a given interval such that its length is minimum.

3.2 T-GQL by Example

The syntax of the language has the typical SELECT- MATCH-WHERE form. The SELECT clause performs a selection over variables defined in the MATCH clause (aliases are allowed). The MATCH clause may contain one or more path patterns (of fixed or variable length) and function calls. The result of the query is a temporal graph. This can be modified by the SNAPSHOT operator, which allows retrieving the state of the graph at a certain point in time. The basic syntax and semantics are introduced using the social network in Figure 2. Path functions implementing the consecutive path semantics are covered using a flight scheduling example. The queries below aim at giving the flavour of the language. A detailed description can be found in [10]. Note that the data structure is transparent to the user who only cares about the semantic time-varying objects rather than Object, Attribute and Value nodes.

Query 1 *Find the continuous paths between Peggy Sue-Jones and Peter Green with a minimum length of two and a maximum length of three.*

```
SELECT paths
MATCH (p1:Person), (p2:Person),
paths = cPath((p1) - [:Friend*2..3] -> (p2))
WHERE p1.Name = 'Peggy Sue-Jones'
      and p2.Name = 'Peter Green'
```

In this query, the cpath function computes the continuous path. The result is only one path of length three (paths of length one are discarded in this query). It can be seen that this starts at Peggy Sue-Taylor and ends at Peter Green, passing through Pauline and Cathy, and it was valid continuously between 2010 and 2017.

Query 2 *Find the names of the persons such that there is a continuous path of length two or three from them to Peter Green.*

```
SELECT p1.Name
MATCH (p1:Person), (p2:Person)
WHERE p2.Name = 'Peter Green'
and cPath((p1) - [:Friend*2..3] -> (p2))
```

As a more involved example, the query below uses the temporal clause WHEN.

Query 3 *Find the friends of Peggy while she was living in Antwerp.*

Peggy lived in Antwerp in the interval [1999-*Now*], thus, any person that was a friend of Mary *at any instant* of that interval would be in the result.

```
SELECT p2.Name as friend_name
MATCH (p1:Person) - [:Friend] -> (p2:Person)
WHERE p1.Name = 'Mary Smith-Taylor'
WHEN
  MATCH (p1) - [e:LivedIn] -> (c:City)
  WHERE c.Name = 'Antwerp'
```

For WHEN queries, the wildcard selection can only be performed on the nodes of the outer query (the MATCH clause). In a nutshell, the inner query returns a collection of intervals, and the WHEN clause performs a BETWEEN operation with these intervals. Only Peter Green is in the result, since Pauline was Mary's friend from 2010 through 2018.

T-GQL also addresses path queries over consecutive paths. To illustrate this, the graph containing flight schedules and airport locations in Figure 4 is used. The following query asks for the earliest-arrival path from Tokyo to Buenos Aires.

Query 4 *How can we go from Tokyo to Buenos Aires as soon as possible?*

Here, the difference with the continuous path semantics is clear: a path in the solution of the latter must be such that the intervals of the edges are pairwise disjoint. The T-GQL query is written as follows:

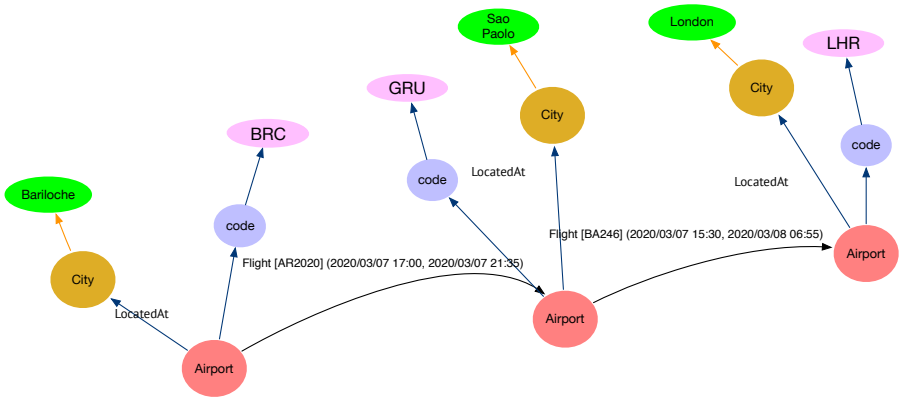

Fig. 4 A temporal model for flight schedules.

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport),
      (c2:City)-[:LocatedAt]->(a2:Airport),
path = fastestPath((a1)-[:Flight*]->(a2))
WHERE c1.Name = 'Buenos Aires' AND
      c2.Name = 'Tokyo'
```

4 A Platform for Graph Visualization

The main goal of this work consists in developing a graph visualization platform that seamlessly integrates with the T-GQL engine presented in [10]. That is, given a T-GQL query, the result (a temporal graph) must be captured by the visualizer, displayed in a panel, and navigated back and forth in time. In addition, the platform must allow visualizing a temporal graph across time, by selecting a date interval and navigating across time using a slider button. The implementation of the temporal graph data model proposed in [10] is based on structural information that allows handling time in a transparent way. The visualization tool must hide this structure from the user, leaving only the relevant nodes and edges to be seen and hiding from the user the underlying data structure. This is described in detail in Section 4.3. Finally, the tool must be able to handle an mount of nodes and edges usually larger than the ones that can fit on a screen.

4.1 Language and Frameworks

The graphic interface is written in Javascript, in order to favor accessing to interface and visualization libraries and bootstrapping the work. The *React.JS*[9] framework is chosen for building the platform and handling user interaction.

[9] `https://es.reactjs.org/`

In React.JS, access to information is handled as follows. In an HTML web page there is only one DOM element[10] that is visually represented, denoted here as the Real DOM. React manages DOM elements through a Virtual DOM hooked to the Real DOM, so the chosen library must be able to adapt to this. In case of failure, conflicts occur at the React's side when trying to synchronize its Virtual DOM with the real one. Well-known visualization libraries and frameworks work with their own Virtual DOM or even modify the real DOM, in order to leverage the creation of visualization elements. The vis.js library is also used, since it is 100% compatible with React and there is a way to avoid component references and only re-render through normal component's lifecycles.

4.2 Platform Interface

To provide a clean interface, a simple layout is developed. Figure 5 depicts the welcome screen. This layout is the same for every view across the whole platform. The section labelled as '1' in the figure is the navigation drawer which contains the different navigation links to switch between the different screens. This navigation drawer is collapsable into single icons in order to provide a larger main section area when preferred. Section '2' is the main panel and aims at holding 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[11], the Neo4j's visualization platform, handles its interface, and, given that the underlying database is Neo4j, users will find this interface familiar. The *Settings View*, depicted in Figure 6, allows setting different configuration parameters. Following Patel and Mistry [18], the focus is on the layout and motion inside each view. Pieces of information are put together into card modules appearing whenever they are required.

After the connection test, there are two groups of 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 to several different databases. All the information needed is retrieved through a direct query to the database (there is also a default selector). For each node type there are four available settings, three of them strictly used for visualization styling purposes and one for querying. Each of this four columns has its own type of button. The Type button allows defining the type of node to be configured. The Color selector allows customizing visualization. The Color Picker has a palette selection, gradient, and also manual RGB color insertion. This feature is important for large graphs, to easily relate similar colors with node types.

Node and Edge Settings. The Icon selector allows the user to choose from a list of preloaded icons to be used in the selection module on the visualization

[10] According to W3.ORG [14] DOM is the acronym for Document Object Model and it is the application programming interface (API) for HTML and XML documents.

[11] `https://neo4j.com/product/bloom/`

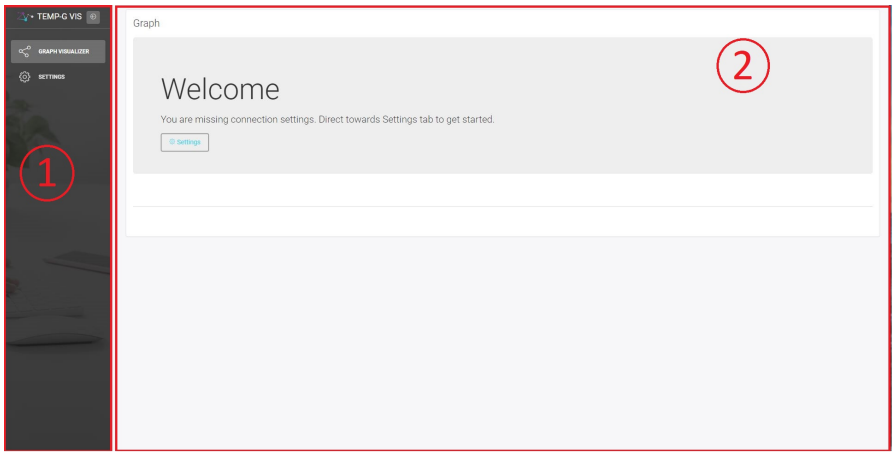

Fig. 5 TGV general layout

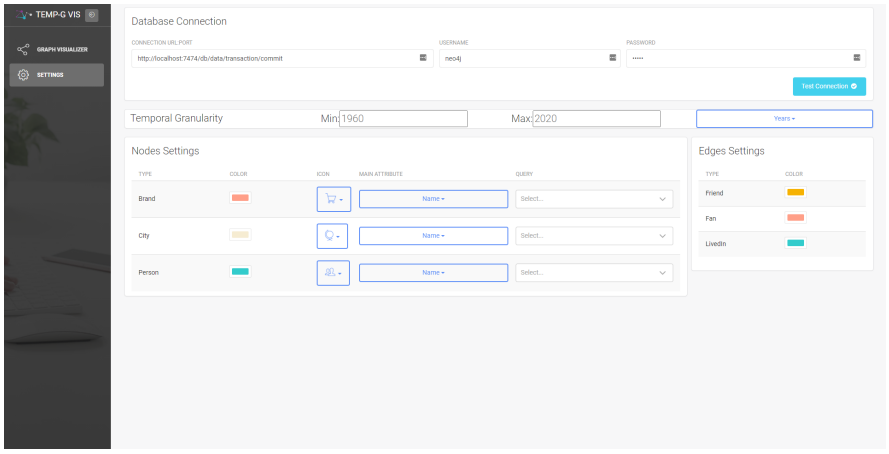

Fig. 6 TGV Settings View

view, to quickly identify the node type. The Main Attribute selector has two main purposes. Due to the temporal property graph structure, every Object node can have several Attribute nodes with their own Value node. Thus, it is important to define which of these properties is the most relevant one, to be used for quick identification of the node in the interface (e.g., the Name property for a Person node). When hovering over a node, the value of this attribute is shown. Note that, due to the number of expected nodes, the visualization is text-free and attributes become visible only when the user places the cursor over a node or edge. Nevertheless, when a query includes a selection condition in the WHERE clause, the nodes satisfying this condition are displayed with a wider border. This is further explained below.

The Query selector is a query builder, which allows starting the navigation focusing the visualization on the part of the temporal graph that is of interest

to the user, for example, filtering out node types the user is not interested in. To build the query, a dropdown list offers a checkbox allowing multiple selections at the same time. Finally, since lists could be large, the component also has a search box that filters the dropdown list through a fuzzy search.[12]

Since T-GQL allows heterogeneous property graphs (that is, graphs with several different relationships between nodes), users can also set the color of each relationship (edge), to easily distinguish between them. The selection component is the same color picker described before for nodes.

Temporality Settings. As it is described below, management and selection of temporal filters is done through a slider component. This slider, depicted in Figure 7, has several characteristics that impact on its usability, depending on the type and amount of data that the user manipulates. A limit selector allows defining the lifespan of the slider, while a granularity selector defines the step in which users can increase or decrease the selection for the sliding filter. Nevertheless, when the step (i.e., granularity) is small, it becomes a continuous slider. Four granularities are allowed, namely years, months, days, and hours.

Fig. 7 Visualizer Slider

The main panel is depicted in Figure 8. Several modules or boxes are defined (indicated with the letters A to D in the figure), according with the functionality they encapsulate. We describe these modules are described next.

- A. Query Box. The query editor, located in the upper part of the main panel. It is implemented as a customization of the *CodeMirror* project.[13]
- B. Graph Module. This is is the main module where the results retrieved by the query are displayed. Figure 8 shows this module and its two parts: the center one displays the temporal graph; the bottom part includes the color references, that provide information for better understanding the visualization. Shape and color are leveraged to easily allow visual association. Colors are used to distinguish types, while shapes allow distinguishing elements, with circles referring to nodes and lines referring to relationships. This practice of taking advantage of n-dimensions of visualization follows the idea of visual variables and their categorizations presented by Bertin [5]. A slider is located below these references (see Figure 7 for a detailed view). The text on the right-hand side of the slider remarks the selected period.

[12] 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 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`

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

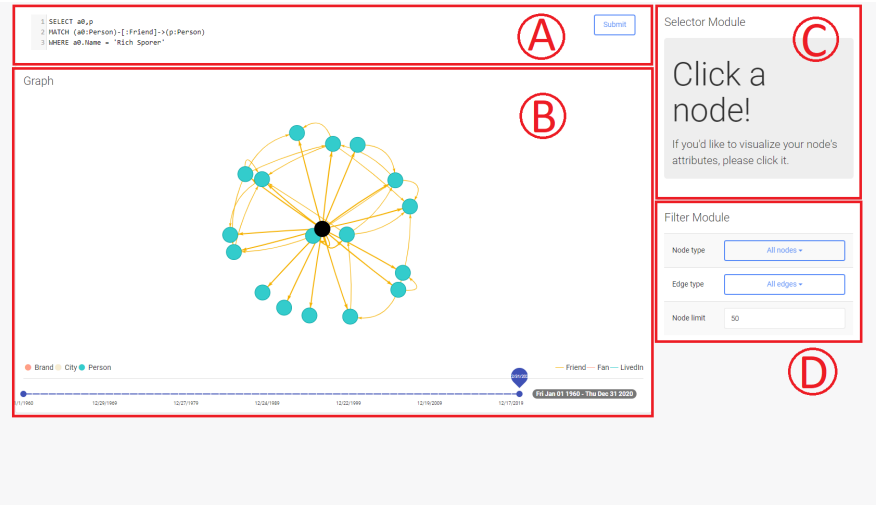

Fig. 8 Temporal Graph Visualizer (TGV)

– C. Selector Module. Located in the upper-right part of the panel, this module provides detail about any node that is selected. Given the large amount of data the temporal graph can contain, the node information is exposed in a separate box for exploring the content. This module displays information about the node, including a list of the different Attribute and Value nodes related. The temporality related to them is added (that is, the historical values and intervals of the node properties), so that behaviour can be better understood when filtering the graph through the slider. This is shown in Section 5.
– D. Filters. In this module, selections can be performed. Although these are simple filters which are applied after the query is run, they are useful to clean and focus the visualization on what really matters to the user.

4.3 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 the user. This structure is shown in Figure 9 for a portion of the underlying Neo4j graph that implements the temporal graph of Figure 2. There are three types of nodes in this structure: Object nodes (in blue), Attribute nodes (in orange), and Value nodes (in green). Note that Person, City, and Brand are Object nodes, while the LivedIn, Friend, and Fan relationships exist between two Object nodes, distinguished through the property title, which can take the values Person, City, and Brand. Object nodes are linked to Attribute nodes, and the latter to Value nodes, through a relationship called Edge. Attribute nodes also contain an attribute title instantiated with the value Name (in this example), and Value nodes contain an attribute value instantiated with the actual value of the attribute, and an attribute interval, containing the interval for this value. That is, in

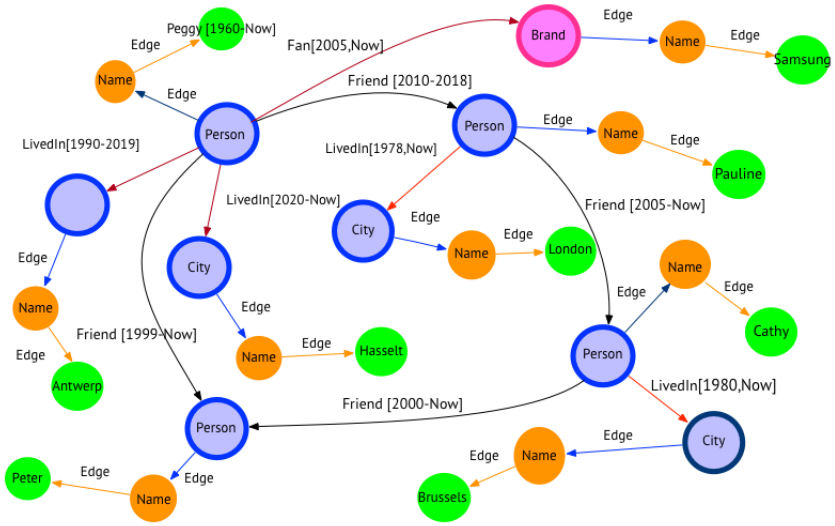

Fig. 9 Social network underlying implementation.

Figure 9, the names in the green nodes are the values of the value property in nodes of type Value, and the names in the blue nodes are the values of attribute title in the nodes of type Object in the underlying graph. Thus, the temporal semantics is implemented through this complex structure which must be hidden from the user for both, querying and visualization. The framework described in this paper builds on this idea. The information conveyed by the schema is displayed in a concise and clear way, coalescing Attribute and Value nodes into the Object nodes, allowing the user to expand the Object nodes when she needs them. This can be seen in Figure 10, which shows different object types and their relationships. All other nodes remain hidden, although their information is still accessible through other methods such as hovering over or selecting the node. To the right of the same figure, a Person node can be seen. Note that all the information about this person (including the validity intervals of the attributes) is available for the Person node over which the cursor is being positioned.

4.4 Handling Temporal Granularity

Handling different *time granularities* is a key issue in temporal databases. T-GQL allows representing temporal objects with different granularities in the same database. Further, the query may include temporal conditions with a granularity different from the one of the databases objects, and T-GQL deals with this in a transparent way. For example, to compare the year 2020 with the day 10-20-2020 both dates must have the same granularity. Thus, a conversion is performed. Although T-GQL handles granularity at database level, the interface must also deal with this issue when handling time visualization con-

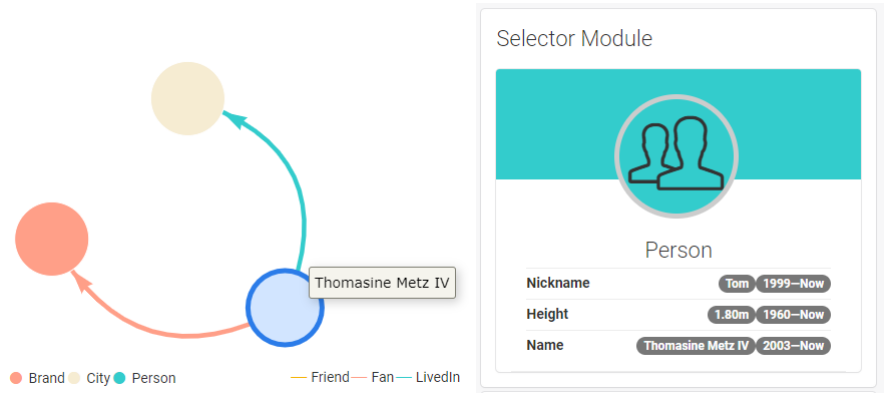

Fig. 10 Object types and relationships (left); A Person Object node.

straints. For time filtering on the visual interface, the only conversions required refer to the time span to be displayed. That is, the minimum value in the slider interval must be converted to the minimum value allowed for the granularity in which data are displayed, and analogously for the maximum. The same technique is used for the extreme points of the intervals of the properties. In a nutshell, there are two intervals, which are are converted into timestamps, and then a simple comparison checks if the temporality of the element overlaps with the temporality of the filter. Finally, temporal databases represent the (moving) current instant with the special value "now" as the upper bound of the intervals. This value is implemented as the largest number allowed by the database system.

4.5 Color Criteria

The mechanism for choosing the color criteria when drawing graphs (specifically paths), is inspired in the work by Cynthia Brewer [7], who developed an online tool for choosing color palettes [14]. There are three types of schemes, namely: sequential, diverging and qualitative. For the present paper, a qualitative palette was selected since it allows distinguishing the different paths, since the colors are more contrasting. Specifically, the chosen palette contains 12 different colors (Figure 11). It is worth noting that when there are more than 12 different paths to show (see Section 4.6), the colors repeat cyclically, meaning that path 13 has the same color as path number 1.

Fig. 11 Twelve-class paired color palette by Cynthia Brewer.

[14] https://colorbrewer2.org/

4.6 Rebuilding the Temporal Graph

Since T-GQL does not always return a graph, the temporal graph must be rebuilt from the query result in order to be displayed on the framework panel. The interaction interface is the integration point with the T-GQL engine. The visual interface software hooks up to the backend through an endpoint, using the database engine as a service provider. This avoids creating interfaces to interact with both applications. The T-GQL engine returns the query results in JSON format, and the visualization interface must take the result and display it graphically, as explained next.

First, the connection to the database is established. If the connection is successful, the user selects her visual preferences, as explained above. Then, in the visualization panel the user writes a T-GQL query, which is submitted to the TBDG service and the rebuilding process starts. When the visualization result is displayed in the main panel, the user can narrow the types and number of nodes to be shown. A temporal filtering can also be applied using the slider component, removing every node and edge outside the selected range. Finally, once the visualization is represented in the main panel, the 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, she can click on any node and the information will be displayed. The following example helps to understand the idea. The query asks for the friends, and the friends-of-friends (FOFs), during a continuous interval, of a person whose name is "Williams Little", together with information of the latter.

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

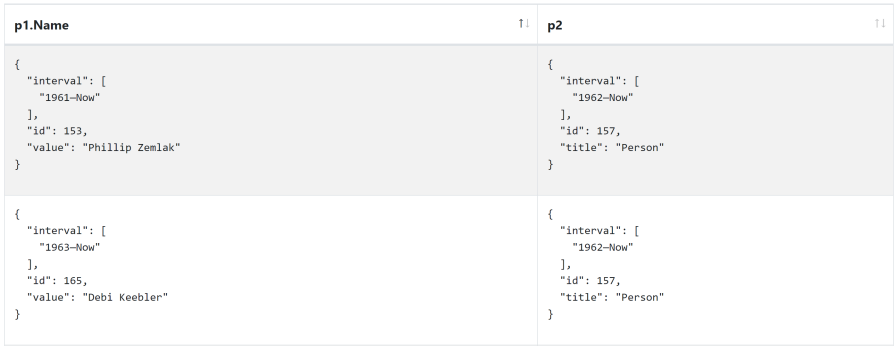

Fig. 12 Response from the database engine in JSON format.

The TDBG service returns the result in JSON format, containing a list of nodes and attributes satisfying the query. A portion of the result is shown in Figure 12. It can be seen that no information on edges is returned, and such

information is needed to display the graph. Additional node information is also required, since the result only contains the attributes requested by the user. Thus, the database must be queried again, in order to get this information. Note, however, that the new query only involves the node in the result of the user query, not over the whole graph. Thus, the identifiers of the nodes in the result are collected, and the query below is submitted:

```
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]
```

If the original query asks for paths, a different mechanism must be applied. The TDBG response returns the nodes in each path in the natural temporal order. The program iterates over the paths and each path is given a different color code. This color code is stored in the node. Therefore, 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. That is, the first path in the iteration has a color code of 0 and the last one a color code of 11. Moreover, every different node encountered is stored in a list, which is used to fetch the edges involved, with the mechanism explained above.

With all these data, iteration through the result nodes is performed, validating the node limit and types from filters. Temporal filtering is then applied by performing an interval check. Then, the color is chosen depending on whether the node is part of a path or not. If it is, the previously chosen color 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). Before consolidating this node in the visualization, a verification is made to corroborate that it belongs to the WHERE clause, in which case it is distinguished, for clarity. After nodes are completed, an iteration over the edges is performed, validating intervals and types from filters.

In case the query returns paths, another filter is applied as follows. A set denoted *restricted edges* contains every edge in the query result, along with its direction. This set is built when the edges are obtained from the database, and it is used to prevent inserting (and displaying) the same edge twice with different directions. Finally, the color is chosen, again taking into account if it is part of a path or not. If it is, the algorithm 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 (which has been selected in the settings view) for that type of edge is used. The graph visualization process is described in Section 5.

4.7 Preprocessing the Query

Finally, it is worth mentioning that, before building the graph, a preprocessing of the query needs to be performed in order to obtain the data needed to

generate the graph. This is explained next. First, attributes from the SELECT clause are captured. For example, the following query:

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

is transformed into:

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

The reason behind this is that the first query would return the list of nodes, but the identifiers of those nodes correspond to the id of the Value nodes. To correctly obtain the edges involved in the query, the ids of the Object nodes are needed. By converting c.Name to c, the engine now returns the ids of the Object nodes and with that list of ids, the edges can be retrieved as explained above. The attributes that the user needs are not relevant in this case, because the displayed graph shows all the attributes in the selector module. This process is done by finding the start of the SELECT clause, the start of the MATCH clause, and then obtaining the sub-string in-between these indices. Then a split is done using the comma as separator. Now every element corresponds to a an item in the SELECT clause. Finally, a dot is looked for when parsing every item. If found, from the dot on, everything is sliced. So every item now has only object and no attributes. The SELECT clause is built again with the new items.

Second, the SELECT clause is parsed, to find all the items requested by the query to highlight them when building the graph. To achieve this, the following two regular expressions 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 requested in the query. These values are stored in a array that is then used to highlight the desired nodes.

5 Querying and Visualizing Temporal Graphs

This section shows how the visualizer is used to navigate and analyze temporal property graphs. The first part shows how the complete graph is navigated over time. For this, a query returning the whole graph in Figure 2 is submitted. In the second part, a query restricting the social network graph is shown, to convey the idea of the usefulness of the tool presented here, based on the assumption that a user normally focuses on the history of just a part of the graph.

Figure 13 shows the social network graph as seen in the visualizer interface. The query in the top panel retrieves all Person, City, and Brand nodes in the graph. The result is displayed in the main panel. The cursor hovers, in this case, over the Friend relationship. It can also be seen that, on the panel on the right-hand side, the Person node corresponding to Peggy Sue-Jones is

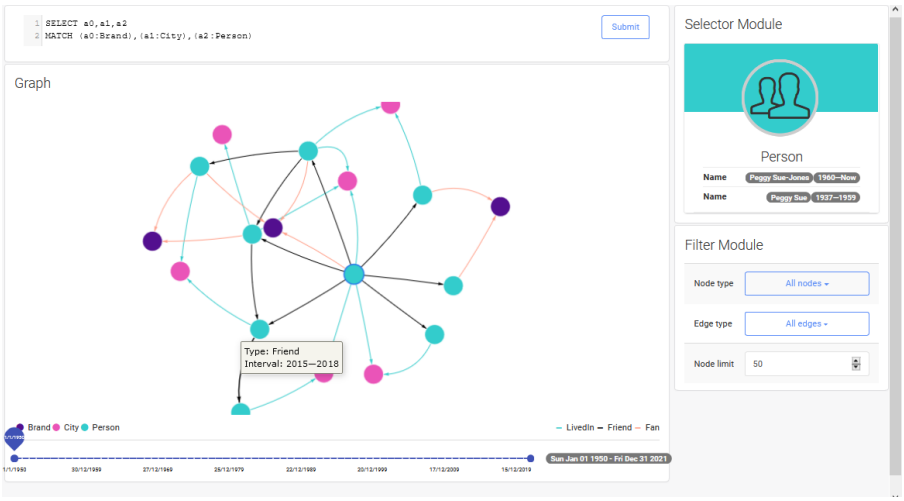

Fig. 13 Social network temporal graph of Figure 2.

displayed, along with the history of her name (as a single and married person). Also, note that the node corresponding to Peggy appears with a thick border in the main panel. Also notice that there are three edges from Peggy's node to the cities where she lived: Brussels, Antwerp, and Hasselt (see Figure 2).

Now, the slider is moved so the graph that is displayed is restricted to the interval 2019-2021. The result is shown in Figure 14). The Friend relationship in the period 2015-2018 is no longer present. Also, since Peggy lived in Brussels between 1979 and 1989, this relationship is no longer either.

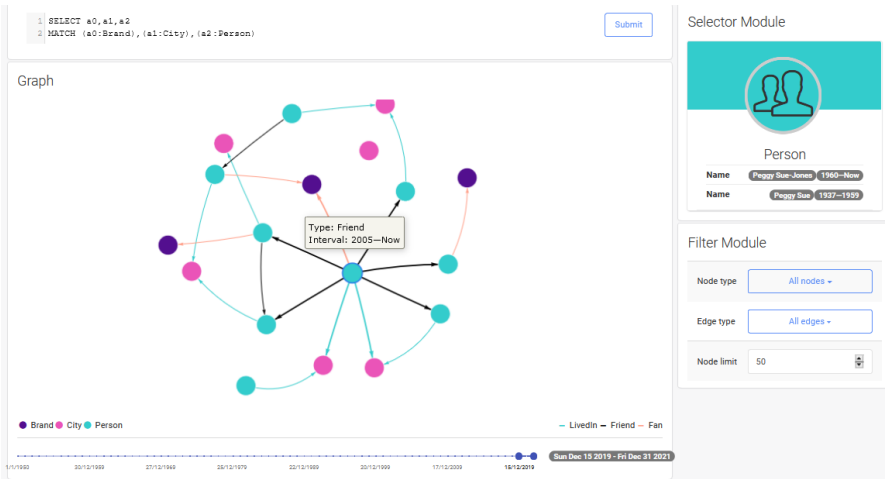

Fig. 14 Social network temporal graph as of the interval 2019-2021.

Finally, Figure 15 shows the complete graph, but hovering over the LivedIn relation corresponding to the period when Peggy lived in Antwerp, and now the selected node is the one representing the city of Antwerp.

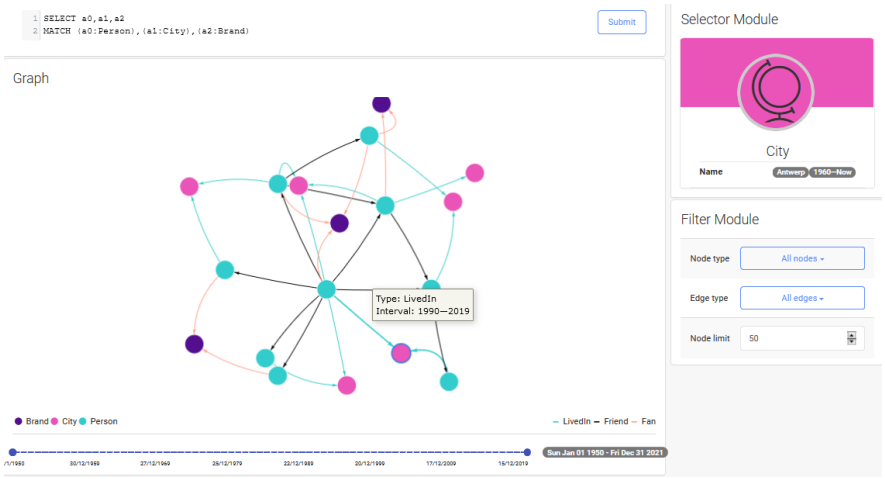

Fig. 15 Social network temporal graph displaying a City node.

Next, the user asks for the friends of Peggy when she was living in Antwerp (that is, between 1990 and 2019). Thus, the used is only interested in seeing the Person nodes corresponding to Peggy and her friends, and the City nodes corresponding to the cities where Peggy's friends lived. Thus, she writes the query below.

```
SELECT p1,p2,c1
MATCH (p1:Person) - [:Friend] -> (p2:Person)-[:LivedIn]->(c1:City)
WHERE p1.Name = 'Peggy Sue-Jones'
WHEN
  MATCH (p1) - [e:LivedIn] -> (c:City)
  WHERE c.Name = 'Antwerp'
```

Note that in this case, the query does not ask for the continuous paths, so any path in the graph will satisfy the query provided it matches the pattern, regardless the time intervals of the Friend relation. Figure 16 depicts the result. Note that now the user poses the cursor over Peter Green, one of Peggy's friends. The node corresponding to Peggy is marked in thick black line (since it is in the WHERE clause), and the user also selected the Person node corresponding to Cathy, another friend of Peggy. This node is marked in green thick line. Also note that the LivedIn relationship for the cities where Peggy lived, are not indicated in the figure.

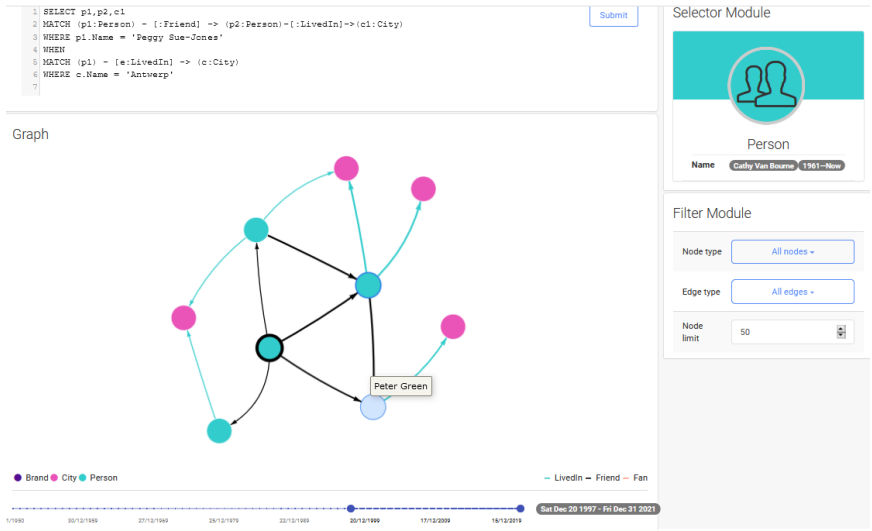

Fig. 16 Peggy's friends when she lived in Antwerp, and the cities they lived in.

6 Experimental Evaluation

A series of experiments were carried out to study the performance of the application under different scenarios. The study consisted in running a collection of queries over different databases with a varying number of nodes and edges. It is important to remark that the experiments reported here are not aimed at evaluating the query response times (which depend on the temporal database engine), but rather to estimate the overhead introduced by the visualizer. Queries over the database are run using the client tool denoted TDBG explained above, and described in [10]. Two cases are studied: (a) A social network similar (but larger) to the example discussed above, to analyze continuous paths; (b) A flight schedule network, to address consecutive paths. All experiments were run under the same environment, a Neo4j 3.5.17 server run on Ubuntu 16.04 64-bits, with a 12 core CPU and 25 GB of RAM.

6.1 Social Network Use Case

Three different (synthetic) social network databases were created with an increasing number of nodes and edges. Table 1 shows the parameters of each database. The database contains three kinds of Object nodes: Persons, Cities, and Brands. There are also relationships: Friend (between person nodes), and Fan (from a person to a brand).

Five different queries, listed below, were used to analyze the social network database.

```
Q0 -    SELECT p, c, n
        MATCH (p:Person),(c:City),(n:Brand)
```

	Social Network 1	Social Network 2	Social Network 3
Nodes	392	1950	2100
Edges	1348	13537	23870
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
cPath min Length	5	5	5

Table 1 Parameters used in the creation of social network databases.

```
Q1 -    SELECT paths
        MATCH (p1:Person), (p2:Person),
        paths = cPath((p1)-[:Friend*2..3]->(p2))
        WHERE p1.Name = 'Hilton Turner' and p2.Name = 'Jan Dickens'

Q2 -    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'


Q3 -    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'

Q4 -    SELECT paths
        MATCH (p1:Person), (p2:Person),
        paths = cPath((p1)-[:Friend*2]->(p2))
        WHERE p1[id] = 250
```

Query Q0 looks for all people, cities and brands in the database. Query Q1 searches for all the continuous path friendships with distance 2 and 3 between Hilton Turner and Jan Dickens. Query Q2 retrieves all the friends of Hilton Turner and the cities where they lived between 2000 and 2004. Query Q3 retrieves all the friends of Hilton Turner when she was living in Danaeview (a fictitious place, of course). Finally, Query Q4 searches for all the friendships of distance 2 from the person with id=250.

Table 2 depicts the difference in the query response times when the queries are run using the T-GQL client and the visual interface, TGV. We remark that, with social networks (SN) 2 and 3, Query Q0 did not finish within a reasonable time, because of the high maxFriendship parameter (25 in this case, while in sample 1 this parameter has a value of 5). Also, response times for Queries Q0 and Q4 are much higher than for the other queries, because of the

Query	Social Network 1	Social Network 2	Social Network 3
Q0	145	N/A	N/A
Q1	38	43	44
Q2	39	60	65
Q3	42	47	77
Q4	46	121	885

Table 2 Difference in response times between TGV and the T-GQL client for the Social Network graph (msecs).

higher number of nodes in the query result. To speed up query performance, temporal indexing methods are being developed at the moment of writing this paper, although this is beyond the scope of this work. It can be seen that for Queries Q1 to Q4, the difference in response times remains almost constant, around 40 milliseconds for SN 1. All these queries return less than 10 nodes in the result. The same happens with Q1 and Q2 in SNs 2 and 3, where the difference stays around 60 milliseconds, and the number of nodes in the result is above 10. Finally, for Query Q0 in SN 1, and Q4 in SNs 2 and 3, the difference is above 100 msecs. It follows that, as expected, as the number of nodes in the result of the query increases, so does the difference between the response times in TGV and TDBG. Moreover, in all cases, the difference is lower than 1 second. Figures 17 and 18 depict the results obtained, with respect to the number of nodes in the result (regardless the queries). The former shows the running times using both tools (note that both curves move together, with a small difference between each other), while the second shows the difference between such times. Figure 18 shows that, below 110 nodes, the line is almost horizontal, strongly increasing after this value.

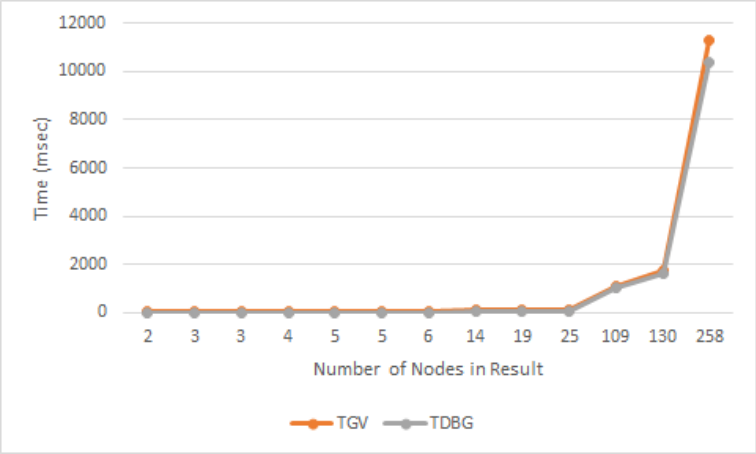

Fig. 17 Response times depending on the number of nodes for the Social Network graph (msecs).

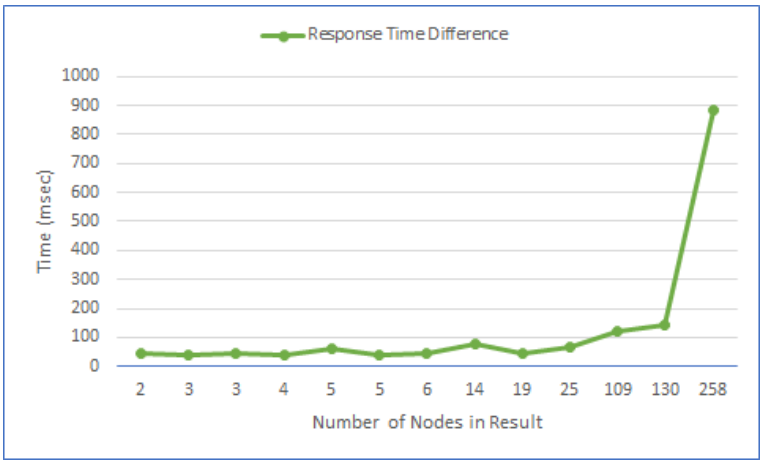

Fig. 18 Difference between the average TGV and TDBG response times depending on the number of nodes for the Social Network graph (msecs).

	Airport 1	Airport 2	Airport 3
Nodes	60	600	1800
Edges	82	788	2400
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 Use Case

To test consecutive paths, a different scenario was devised. In this case, three airport databases were created with an increasing number of nodes and edges. Table 3 shows the databases characteristics. The graph contains two kinds of nodes, Cities and Airports, a relation LocatedAt, from airports to cities, and another relation Flights, between airport nodes. Figure 19 shows the complete graph loaded on the visualizer. The queries are listed next.

```
QA0 -   SELECT a, c
        MATCH (a:Airport), (c:City)

QA1 -   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'

QA2 -   SELECT path
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport),
        (c2:City)<-[:LocatedAt]-(a2:Airport),
        path = latestDeparturePath((a1)-[:Flight*]->(a2),
```

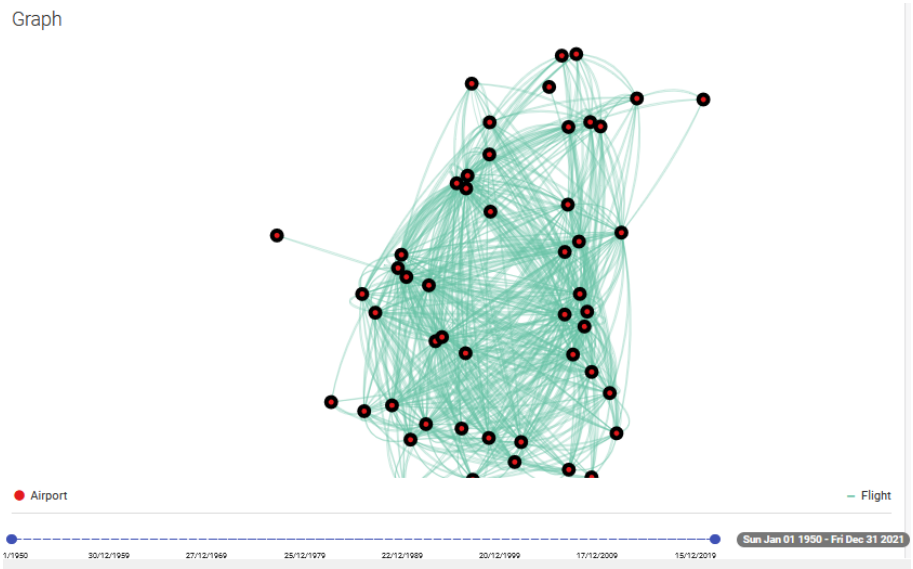

Fig. 19 Airports temporal graph

```
        '2020-12-11 17:04')
        WHERE c1.Name = 'Port Berniecebury'
        AND c2.Name='West Hipolitohaven'

QA3 -   SELECT a1, a2, c1
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport)-[:Flight]
        ->(a2:Airport)
        WHERE c1.Name = 'Port Berniecebury'

QA4 -   SELECT a1, a2, c1
        MATCH (c1:City)<-[:LocatedAt]-(a1:Airport)-[:Flight*4]
        ->(a2:Airport)
        WHERE c1.Name = 'Port Berniecebury'
```

Query QA0 lists all the airports and cities in the database. Query QA1 computes the fastest paths between the airports located in the cities of Port Berniecebury and West Hipolitohaven. Query QA2 searches for the latest departure path since 17:04 in 2020-12-11 between the previously mentioned cities. Query QA3 retrieves all the airports and cities that have direct flights from Port Berniecebury. Finally, Query QA4 looks for all the airports and cities at a 4-scale distance from Port Berniecebury. Of course of the city names are fictitious.

Table 4 shows the difference in response times obtained running the queries over the visual interface and the T-GQL server. Figures 20 and 21 show, like above, this difference with respect to the number of nodes in the result, regardless the query. As in the social network case, with less than 10 nodes in the query result, the response times are quite constant, around 20 milliseconds

Query	Airport 1	Airport 2	Airport 3
QA0	22	66	130
QA1	18	25	17
QA2	22	14	14
QA3	20	16	14
QA4	21	18	14

Table 4 Difference in response times between the interface and the T-GQL server for the Airports graph (msecs).

in this case. As the number of nodes in the result increases, the difference in response times does too, although the increase curve is less steep. Further, in this case, the maximum difference between response times is about 130 milliseconds for 600 nodes. In the case of Query QA0, with 600 nodes in the result (for the Airport 3 case), the difference is 130 msecs, with 600 nodes in the result, and the total running time to display the resulting graph is about 3 seconds, as shown in Figure 21. This figure also shows that, compared with the social network case, below 100 nodes in the result the line is less straight, although the difference between values is still low.

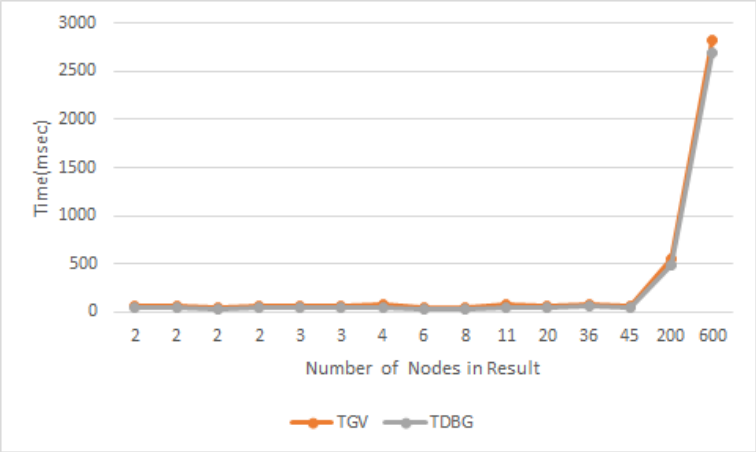

Fig. 20 Response times depending on the number of nodes, for the Airports graph (msecs).

7 Conclusion

We presented a framework for temporal property graphs visualization, denoted TGV, to be used together with T-GQL, a data model and query language for temporal graphs implemented over a Neo4j graph database, and described in previous work. TGV allows editing and running T-GQL queries, displaying the result, and navigating such result across time. Although the visualization research and developers community has produced tools that show the evolution

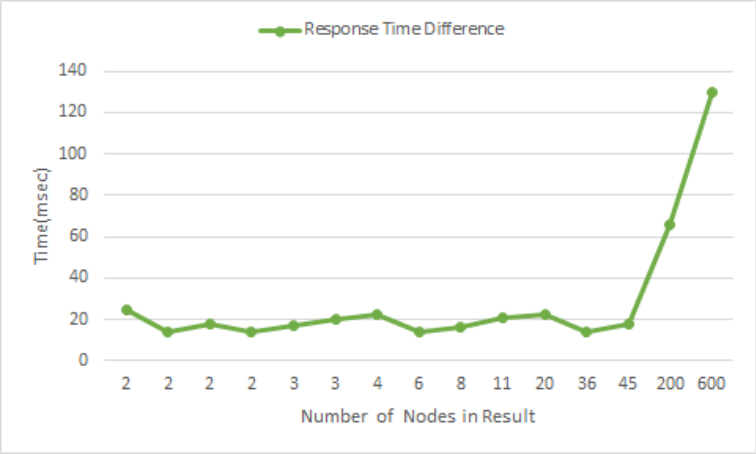

Fig. 21 Difference between the average TVG and TDBG response times depending on the number of nodes, for the Airports graph (msecs).

of graphs over time, to the best of the authors' knowledge none of those tools are based on queries on temporal property graph databases, allowing temporal graph visualizations on-the-fly, based on database query results. The paper describes the visual interface and the different criteria adopted during its development, as well as implementation details. Particular attention was given to the process of hiding from the user the underlying structure of the temporal graph. The results showed that the overhead due to the visualization tool is not relevant, and most of the times negligible.

8 Funding and/or Conflicts of interests/Competing interests

Alejandro Vaisman was partially supported by Project PICT 2017-1054, from the Argentinian Scientific Agency.

All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

1. Angles, R.: A Comparison of Current Graph Database Models. In: Proceedings of ICDE Workshops, pp. 171–177. Arlington, VA, USA (2012)
2. Angles, R.: 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*, vol. 2100. CEUR-WS.org (2018)
3. Bar, M., Neta, M.: Humans prefer curved visual objects. Psychological Science **17**(8), 645–648 (2006). DOI 10.1111/j.1467-9280.2006.01759.x
4. Bassili, J.N.: Facial motion in the perception of faces and of emotional expression. Journal of Experimental Psychology: Human Perception and Performance **4**(3), 373–379 (1978). DOI 10.1037/0096-1523.4.3.373

5. Bertin, J.: Semiology of Graphics. University of Wisconsin Press (1983)
6. Bostock, M.: Force-directed graphs. https://observablehq.com/@d3/force-directed-graph?collection=@d3/d3-force (2017)
7. Brewer, C.A.: Color research applications in mapping and visualization. In: The Twelfth Color Imaging Conference: Color Science and Engineering Systems, Technologies, Applications, CIC 2004, Scottsdale, Arizona, USA, November 9-12, 2004, pp. 1–3. IS&T - The Society for Imaging Science and Technology (2004)
8. Byun, J., Woo, S., Kim, D.: Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. IEEE Trans. Knowl. Data Eng. **32**(3), 424–437 (2020). DOI 10.1109/TKDE.2019.2891565. URL https://doi.org/10.1109/TKDE.2019.2891565
9. Card, S.K., Mackinlay, J.D., Shneiderman, B. (eds.): Readings in Information Visualization: Using Vision to Think. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
10. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. The VLDB Journal (2021). DOI 10.1007/s00778-021-00675-4
11. Hartig, O.: Reconciliation of RDF* and property graphs. CoRR **abs/1409.3288** (2014). URL http://arxiv.org/abs/1409.3288
12. Huo, W., Tsotras, V.J.: 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, pp. 38:1–38:4 (2014)
13. Intelligence, C.: The regraph toolkit. https://cambridge-intelligence.com/regraph/ (2021)
14. Jonathan Robie, T.R.: Rec-dom-level-1. https://www.w3.org/TR/REC-DOM-Level-1/introduction.html (1998)
15. Kreitzberg, C.: The intuitive interface. Princeton University, https://ux.princeton.edu/learn-ux/blog/intuitive-interface (2017). Online; accessed May 15th, 2021
16. Lima, M.: Information visualization manifesto. Visual Complexity, http://www.visualcomplexity.com/vc/blog/?p=644 (2009)
17. Lima, M.: The Book of Circles: Visualizing Spheres of Knowledge. Princeton Architectural Press, NY, NY (2017)
18. Patel, P., Mistry, S.: A guide to material design, a modern software design language. Open Source for You pp. 64–66 (2016). https://www.opensourceforu.com/2016/05/a-guide-to-material-design-a-modern-software-design-language/
19. Rizzolo, F., Vaisman, A.: Temporal XML: Modeling, indexing, and query processing. VLDB Journal **1179–1212**(5), 39–65 (2008)
20. Robinson, I., Webber, J., Eifrém, E.: Graph Databases. O'Reilly Media (2013)
21. Semertzidis, K., Pitoura, E.: Top-k durable graph pattern queries on temporal graphs. IEEE Trans. Knowl. Data Eng. **31**(1), 181–194 (2019)
22. Tansel, A., Clifford, J., Gadia (eds.), S.: Temporal Databases: Theory, Design and Implementation. Benjamin/Cummings (1993)
23. Vartanian, O., Navarrete, G., Chatterjee, A., Fich, L.B., Leder, H., Modroño, C., Nadal, M., Rostrup, N., Skov, M.: Impact of contour on aesthetic judgments and approach-avoidance decisions in architecture. Proceedings of the National Academy of Sciences **110**, 10446 – 10453 (2013). DOI 10.1073/pnas.1301227110