Information Systems Frontiers (2024) 26:1543–1564
https://doi.org/10.1007/s10796-023-10426-1

TGV: A Visualization Tool for Temporal Property Graph Databases

Diego Orlando[1] · Joaquín Ormachea[1] · Valeria Soliani[2] · Alejandro Ariel Vaisman[1]

Accepted: 31 July 2023 / Published online: 15 August 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

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 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 (Angles, 2018; Hartig, 2014; Robinson et al., 2013) have been increasingly gaining popularity, especially for modeling and analyzing different kinds of networks. In short, property 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 (Angles, 2012). 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

Diego Orlando, Joaquín Ormachea, Valeria Soliani and Alejandro Ariel Vaisman contributed equally to this manuscript.

✉ Alejandro Ariel Vaisman
avaisman@itba.edu.ar

Diego Orlando
dorlando@itba.edu.ar

Joaquín Ormachea
jormachea@itba.edu.ar

Valeria Soliani
vsoliani@itba.edu.ar; valeria.soliani@uhasselt.be

1 Department of Information Engineering, Instituto Tecnológico de Buenos Aires Lavardén 315, C1437FBG Ciudad Autónoma de Buenos Aires, Argentina

2 Department of Information Engineering, Instituto Tecnológico de Buenos Aires and Hasselt University, Hasselt, Belgium

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

[2] http://janusgraph.org/

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. (2021) 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 (Tansel et al., 1993) 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 allow defining several different path semantics. In this model, 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 after the arrival to Rome, there is a flight from Rome to Athens, then there is a consecutive path [Paris, Rome, Athens]). Consecutive paths can be of different kinds: shortest, earliest-arrival paths, latest-departure paths, fastest paths, among other ones (Byun et al., 2020). 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. Further, the user is unaware of this structure.

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 data and the level of abstraction present in temporal graph databases. The present paper addresses this problem, building from the concepts and implementation developed in Debrouvier et al. (2021), where T-GQL queries are executed through a client tool called TGDB. The paper describes the design and implementation of a framework, denoted *Temporal Graph Visualizer* (TGV), for interacting with, and visualizing temporal graphs. We approach the problem of visualizing temporal property graphs in two ways. The first one focuses on the platform infrastructure and the second one on temporal graph visualization. The work aims at providing a visualization platform such that the underlying data structure that supports temporal data, remains transparent to the user. This is aligned with another goal of this work, that is, developing a friendly interface for user interaction. According to Kreitzberg (2017), 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. In the platform presented here, users can write their own queries in T-GQL, submit them to the temporal database engine (TGBD), and navigate the result across time in a friendly and interactive way.

1.1 Contributions

This paper describes the design and implementation of a visualization platform for temporal property graphs. The platform capabilities come in two flavours: On the one hand, starting from the whole graph, a slider bar allows navigating across time. This can be awkward for the user, given the amount of data normally present in the graph. Thus, certain tools that prune the initial graph are also provided. On the other hand, the user can write T-GQL queries over an editor panel and the result will be displayed graphically, in a way that she can navigate the graph across time, starting from the query result. For example, if a query 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, also interacting with different graph objects.

We would like to remark the contributions of this work and the differences against other graph visualization tools. To begin with, although there are applications that allow displaying graphs across time (for a comprehensive description see 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. Further, consider Neo4j's front end, which comes with the graph database. This tool allows writing and submitting a (*non-temporal*) Cypher (Neo4j's high-level query language) query, displaying the result in the visualization panel. If the query returns a graph, the result can be displayed graphically (other options are available). If not, it returns a table. In both cases, no further action can be taken over the result. This is a crucial difference with our proposal, which allows navigating the result of a temporal query across time (e.g., through a slider),

even when the query does not return a graph. The mechanism allowing this, are explained in depth in Section 5.

This paper (a) describes the visualization platform introduced above, as well as the rationale for its design: (b) presents the implementation details of the tool, 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; (c) gives examples of the use of the visual interface, either starting from the whole graph or from T-GQL queries; and (d) 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.

1.2 Paper Organization

This paper 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, while Section 5 explains the implementation technical details and how the temporal graph is built after a T-GQL query is submitted and the result returned. Section 6 shows how a temporal graph is queried and the results are displayed, including how navigation across time is performed based on the query results. Section 7 reports the experiments carried out over the prototype and their results. Section 8 concludes the paper.

2 Related Work

This section reviews existing work about the three topics addressed in this paper, namely graph and information visualization, and temporal databases.

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. (1999) define information visualization as the use of computer-supported, interactive, visual representations of abstract data to amplify cognition. According to Lima (2017), 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 (2006), which reveals a strong human preference for curved objects and typefaces. Also, in Vartanian et al. (2013), a similar inclination in architectural spaces was reported.
2. Circles equal happiness: This theory is explained by an experiment by Bassili in 1978 Bassili (1978), 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 circular things.

The so-called “Information Visualization Manifesto” (Lima, 2009) 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) (Bostock, 2017) which allows visualizing a temporal network that changes

[3] https://observablehq.com/

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

over time, using Observable. Most of the use cases in this notebook use a slider, which allows displaying the progression of the graph, together 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, 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. However, no manipulation if the result is allowed (and it does not account for queries not returning graphs as a result).

One of the most recent developments for graph visualization from Cambridge Intelligence, ReGraph (Intelligence, 2021), is a full plain graph visualization tool designed 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.

2.3 Temporal Databases

Temporal databases have long been studied in the literature (Tansel et al., 1993) and they still capture the interest of researchers and practitioners (Gamper et al., 2022). The two main approaches for extending relational databases with temporal semantics are tuple timestamping and attribute timestamping (Clifford & Tansel, 1985; Gadia, 1988). In a nutshell, tuple timestamping yields first normal form relations, while attribute timestamping produces non-first normal form relations. In the model presented in this work, the classic approach of tuple timestamping has been followed. Another discussion in the temporal database field refers to the representation of time, which can be point based or interval based. This has been studied in Toman (1996), and later, a point-based temporal extension to SQL was proposed by the same author Toman (1997). In practice, the interval-based approach is easier to implement and understand, and this has been the approach in practice so far. Finally, one of the most discussed topics in temporal databases refers to temporal dimensions. The time when a data item is valid in the real world is called *valid time*, and it is the one supported by the model in this paper. The time when a data item is valid in the database is called *transaction time*. When a model supports both, it is denoted *bitemporal*. Many other timelines have been defined in the literature, we omit them since they are beyond the scope of this paper. Jensen et al. (1994) proposed a temporal conceptual data model that attempts to capture the temporal semantics of data in a unified way, timestamping the tuples of relations with sets of two-dimensional so-called chronons, thus supporting valid and transaction time.

A large number of temporal data models and query languages have been introduced since the early 1990s. As a result, the TSQL2 (Snodgrass, 1995) standard was proposed as a temporal extension to the SQL-92 standard, supporting valid, transaction, and bitemporal times. Three distinct timelines, namely valid time, transaction time, and user-defined times are supported in TSQL2. The SQL-2011 standard (Kulkarni & Michels, 2012) includes some temporal features, like a period data type, temporal primary keys, temporal referential integrity constraints, temporal predicates, valid-time (application-time), transaction-time (system-time) and bitemporal (system-application-time) tables, as well as temporal insertions, updates, and deletions. In spite of the above, database practitioners are still using standard SQL for manipulating time-varying information. Therefore, Snodgrass (2000) showed how most relational operations can be written in standard SQL. Zimányi (2006) then showed how to implement temporal aggregates and temporal universal quantifiers using standard SQL. As mentioned, the interest in temporal databases is rising again. For example, the work by Dignös et al. (2016) deals with sequenced temporal queries, namely queries that are evaluated at each time point. The driving idea is to reduce a temporal query to non-temporal operators. Also, Lu et al. (2019) proposed a temporal extension to the distributed Tencent database management system for MySQL, called TDSQL. Another example is MobilityDB (Zimányi et al., 2020), a spatiotemporal database management system, that builds on PostGIS (the spatial extension of PostgreSQL), and which extends the type system of PostgreSQL and PostGIS with abstract data types (ADTs), in order to represent spatiotemporal data. Gao et al. (2021) address the problem of solving so-called temporal keyword queries, defined as a query over a temporal relational database, that includes a collection of keywords, a group-by condition along with an aggregate function, and a time condition. Finally, in Grandi et al. (2022), the authors

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

[6] https://visjs.org/

[7] https://reactjs.org/

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

propose a query language and an algebra that integrates in the same framework, streaming, temporal, and classic relational databases.

2.4 Temporal Graph Models

Temporal graphs represent the history of a graph. They can be classified as duration-labeled, interval-labeled and snapshot-based. 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, defined in Section 2.3, is considered in the remainder. In Snapshot-based temporal graphs (Semertzidis & Pitoura, 2019; Huo & Tsotras, 2014), 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. (2021), 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 (title, interval). The title 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 (value, interval) pair. Value nodes contain the actual value of an attribute or property, in a certain interval.

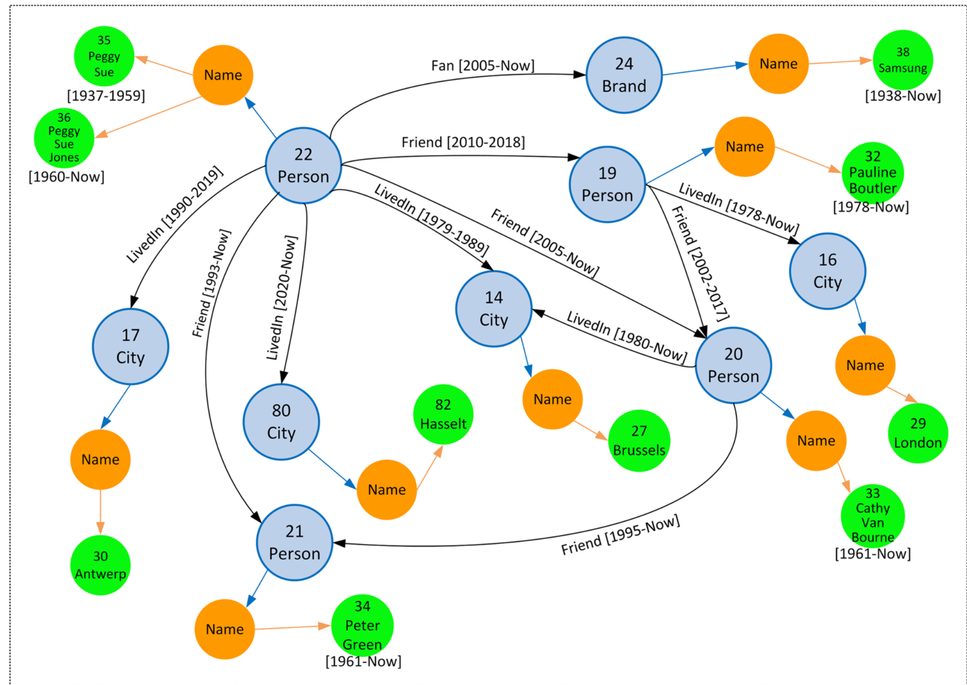
Figure 1 shows a social network represented in the temporal model introduced above. There are three kinds of Object nodes, namely Person, City, and Brand. There are also three temporal relationships: LivedIn, Friend, and Fan. The first one is labeled with the periods when someone lived somewhere. The 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= 35 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, temporal labels are omitted for Object and Attribute nodes.

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.

Continuous paths are paths that are valid continuously during a certain interval (as introduced in Rizzolo and Vaisman (2008)). An example is given in Fig. 2, where two continuous paths can be observed, $(n_1, n_2, n_3, n_4, \ friend, [2, 3])$ 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 (Byun et al., 2020). For example, the *earliest-arrival path* is the path that can be completed in a given interval such that the ending time of the path is minimum, and the *latest-departure path* is the path that can be completed in a given interval such that the starting time of the path is maximum.

Fig. 1 Friend-of-a-Friend Temporal Graph



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 is a Cypher expression, probably extended with some library functions that compute the different kinds of paths explained above. This 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. Below we explain the basic syntax through two examples: the social network in Fig. 1, and a flight scheduling example using the graph in Fig. 3. We remark that the data structure remains transparent to the user who only cares about the semantic time-varying objects rather than Object, Attribute and Value nodes.

As a first example, consider the query "*Find the continuous paths between Peggy Sue-Jones and Peter Green with a length of two.*" The query is written in T-GQL as:

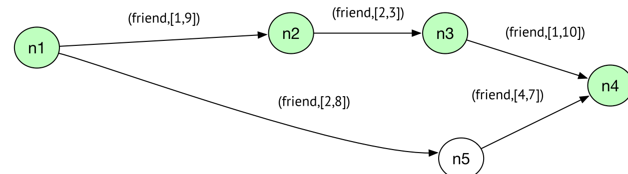


Fig. 2 Continuous paths [cf. Debrouvier et al. (2021)]

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

In this query, the cpath function computes the continuous path. The result contains just one path of length two. It can be seen that this starts at Peggy Sue-Jones and ends at Peter Green, passing through Cathy Van Bourne, and it was valid continuously between 2006 and 2023 (*Now*).

The next query is more involved, and gives an idea of the power of the language: "*Find the friends of Peggy while she was living in Hasselt.*" Since Peggy lived in Hasselt during the interval [2020-*Now*], any person that was a friend of Peggy *at any instant* of that interval would be in the result. This is expressed in T-GQL as:

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

Here, the inner query returns a collection of intervals, and the WHEN clause performs a BETWEEN operation of each interval in the result of the outer query against those intervals. Only Peter Green and Cathy are in the result, since Pauline was Peggy's friend between 2010 and 2018.

T-GQL also addresses path queries over consecutive paths. To illustrate this, the graph containing flight schedules and airport locations in Fig. 3 is used. Thus, we can ask "*How can we go from Buenos Aires to London as soon as possible?*".

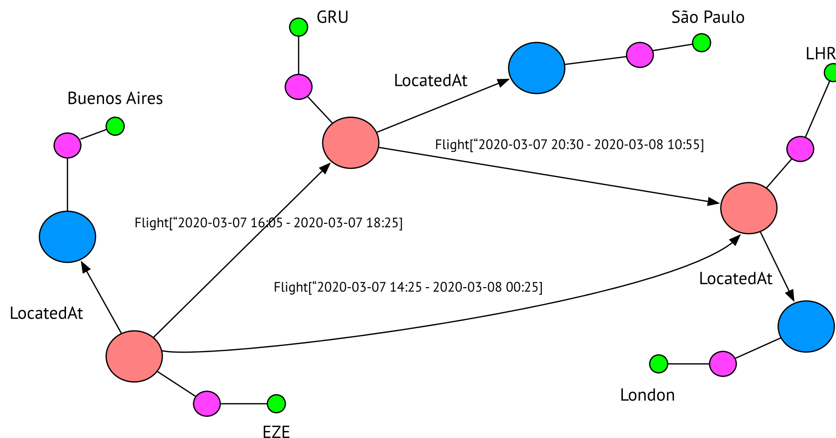


Fig. 3 A temporal model for flight schedules

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. In particular, this is a fastest path query. The T-GQL query is written as follows:

```
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 = 'London'
```

It can be seen on Fig. 3, that the fastest way to go from Buenos Aires to London is taking the direct flight between EZE and LHR. The travel time in this case will be 10 hours. The other possible way, the one that goes through Sao Paulo, takes more time.

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. That is, given a T-GQL query, the result (a temporal graph) is captured by the visualizer, displayed in a panel, and navigated back and forth in time. In addition, the platform allows 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 is based on structural information that allows handling time in a transparent way. The visualization tool hides this structure from the user, leaving only the relevant nodes and edges. Finally, the tool must be able to handle a number of nodes and edges usually larger than the ones that can fit on a screen.

In this section we focus on the graphic interface, and in Section 5 we explain how it is implemented. Section 4.1 presents the languages and frameworks used in the implementation while in Section 4.2 we describe the platform interface. Section 4.3 explains the color criteria chosen for drawing the graphs.

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. 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.

4.2 Platform Interface: Description and Rationale

To provide a clean interface, we developed a simple layout, which remains the same for every screen across the whole platform. It is composed of a navigation drawer on the left-hand side and a main panel. The former contains the different navigation links allowing switching between screens. This navigation drawer is collapsable into single icons in order to

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

[10] According to W3.ORG (Jonathan Robie, 1998) DOM is the acronym for Document Object Model and it is the application programming interface (API) for HTML and XML documents.

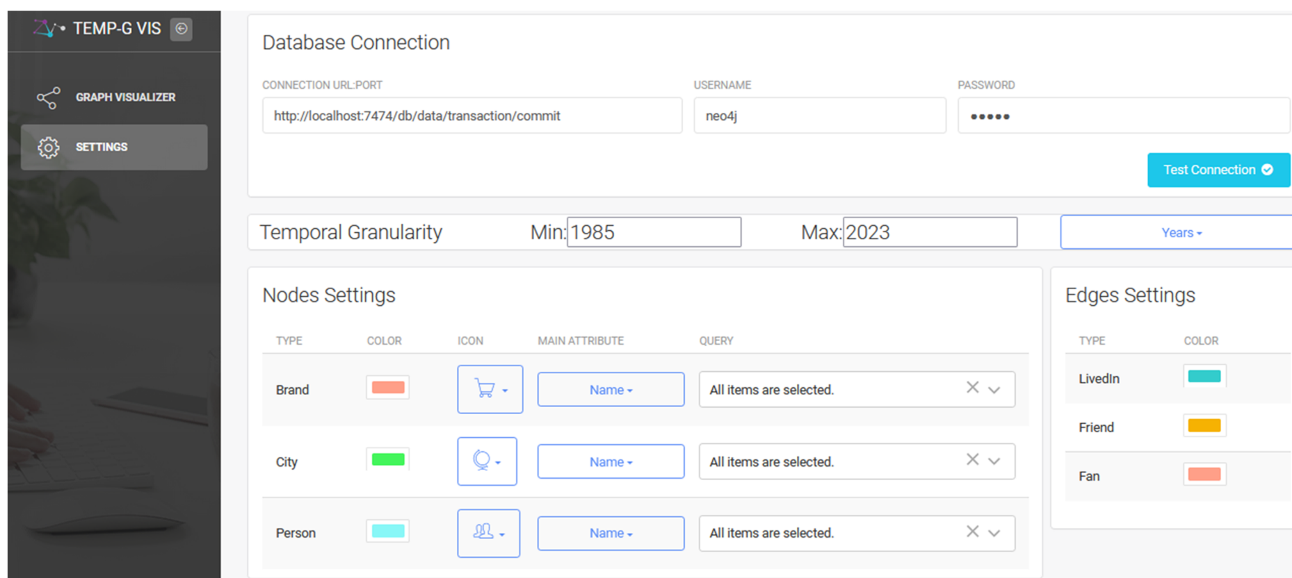


Fig. 4 TGV Settings View

provide a larger main section area when preferred. The main panel aims at holding the varying information regarding the different tabs. With this fixed layout the user can focus on the information in the main section. This approach 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.

Figure 4 depicts the *Settings View*, where we can also see the navigation drawer and the main panel. This screen allows setting different configuration parameters. Following Patel and Mistry (2016), 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. We can see two groups of settings, one for nodes (on the left) and one for edges (on the right). 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. Note that these types represent the way in which user perceives the nodes in the graph, but *behind the scenes they are actually Object nodes in the underlying temporal graph structure.* 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 view, to quickly identify the node type. The Main Attribute selector has two main purposes. On the one hand, when showing for example, a Person node, the user may want to quickly identify the persons being represented. However, such a node may have many attributes describing it. Therefore, it would be of help for the user to define the attribute she would like to use to identify a node at first glance. In the social network case, for example, the user would typically choose the Name property. In the visual interface, when hovering over a node, the value of this attribute is shown. We remark, again, that in the underlying structure, Person is actually an Object node and Name is an Attribute node, but all of these is hidden from the user.

Note that, due to the number of expected nodes, the graph 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 in the next section. 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

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

could be large, the component also has a search box that filters the dropdown list through a fuzzy search.[12]

Finally, 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 Management and selection of temporal filters is done through a slider component. This slider 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.

Main Panel The main panel is depicted in Fig. 5. 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 next.

- A. Query Box. The query editor, located in the upper part of the main panel, is implemented as a customization of the *CodeMirror* project.[13] Here the user writes the T-GQL query that is dynamically displayed in module B. We would like to remark again, that this feature distinguish this tool from the usual temporal graph visualizers (described in Section 2), which require ad-hoc data preparation for particular cases. Also, note that, different from graph visualizers like the Neo4j interface (which manage *non-temporal* graphs), T-GQL and TGV handle *temporal graphs* seamlessly.
- B. Graph Module. This is the main module where results retrieved by the query are displayed. Figure 5 shows this module and its two parts: the center one displays the temporal graph; the bottom part includes the color references (in this figure, the colors for Brand, City and Person nodes, and Friend, Fan and LivedIn relationships), 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 (1983). The time slider is located below these references. The text on the right-hand side of the slider remarks the selected period.
- 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's information is exposed in a separate box for exploring its content. This information includes a list of the different Attribute and Value nodes related. The temporality related to them is also 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 further explained in Section 6.
- 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 Color Criteria

We conclude this section explaining the mechanism for choosing the color criteria when drawing graphs (specifically paths). This is inspired in the work by Cynthia Brewer (2004), who developed an online tool for choosing color palettes.[14] There are three types of schemes, namely: sequential, diverging and qualitative. In the present paper, a qualitative palette was selected since it allows distinguishing the different paths, since the colors are more contrasting. The chosen palette contains 12 different colors (Fig. 6). It is worth noting that when there are more than 12 different paths to show (see Section 5.3), the colors repeat cyclically, meaning that path 13 has the same color as path number 1.

5 TGV implementation

We now explain the technical details of the implementation, with focus on how the temporal graph infrastructure is defined and how query results are processed. In Section 5.1 we explain the abstraction developed to hide from the user the actual implementation of the temporal graph. Section 5.2

[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/

[14] https://colorbrewer2.org/

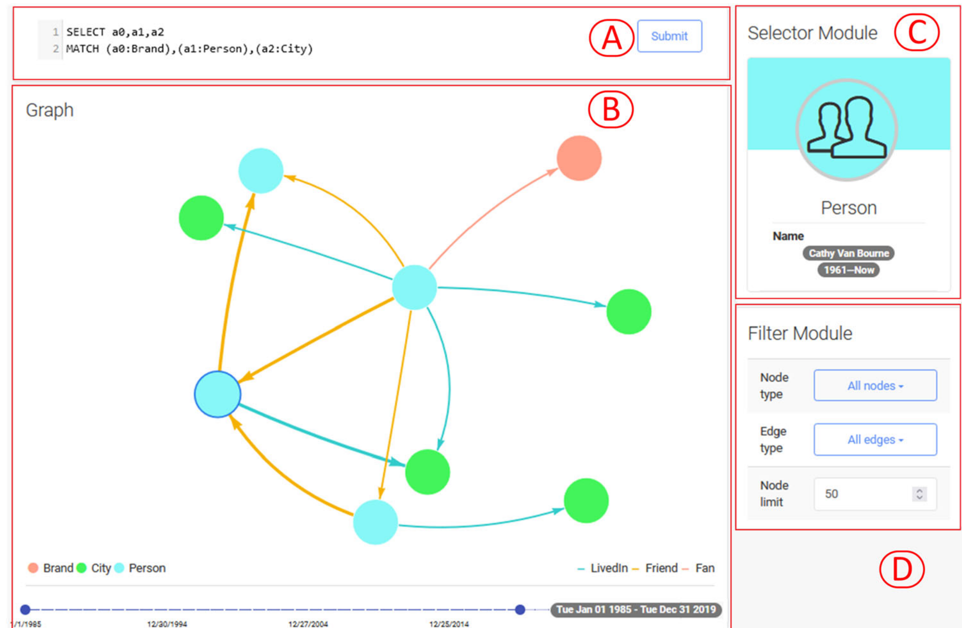


Fig. 5 Temporal Graph Visualizer (TGV)

discusses granularity issues, typical in temporal database environments. Finally, Section 5.3 details how the graph is rebuilt from the query result.

5.1 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. We explain this next.

Consider for example the graph in Fig. 1. The Person node with id = 19, corresponds to Pauline Boutler, who lived in London from 1978 until the present day. Figure 7 shows the Neo4j implementation of the portion of the temporal graph presented in Fig. 1 corresponding to Pauline. As explained in Section 3, there are three types of nodes in this structure: Object nodes (in blue), Attribute nodes (in orange), and Value nodes (in green). The Person, City and Brand nodes are Object nodes. In Neo4j, the labels of the node types are preceded by a colon, as can be seen in the figure. Further, we can see in the figure that Object nodes contain a property called title. In Fig. 7 the title property takes the values Person, City, and Brand. Object nodes are linked to Attribute nodes and Attribute nodes to Value nodes through a relationship called Edge. Attribute nodes also contain an attribute title instantiated in this example with the value Name, which is the name the user has defined for the attribute in all the node types (e.g., the name of a city, the name of a brand, and the name of a person). Value nodes contain an attribute value instantiated with the actual value of the attribute, and an attribute interval, containing the interval for this value (e.g., value = *London*, and interval = $[0, Now]$. In other words, in Fig. 7, the names in the green nodes are the values of the property called value 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. Finally, the LivedIn, Friend, and Fan relationships exist between two Object nodes, with their corresponding intervals. In simple words, we can say that the temporal property graph is represented by a non-temporal Neo4j property graph.

The temporal graph semantics is implemented through the complex structure explained above, which, of course, must be hidden from the user for both, querying and visualization. As an example, Fig. 8a shows how the graph of Fig. 7 is seen by the user when she asks for a snapshot as of 2005, while Fig. 8b shows the snapshot as of 2015. We can see that the user is not aware of the different kinds of nodes, and only sees the abstraction of interest. Instead, in the implementation, 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, for example to display the attributes of a Person node. This can be seen in Fig. 9 which depicts how the user sees the information in the interface. The user just sees a node corresponding to Pauline, and her temporal relationships. All other nodes remain hidden, although their information is still accessible through other methods such as

Fig. 6 Twelve-class paired color palette by Cynthia Brewer

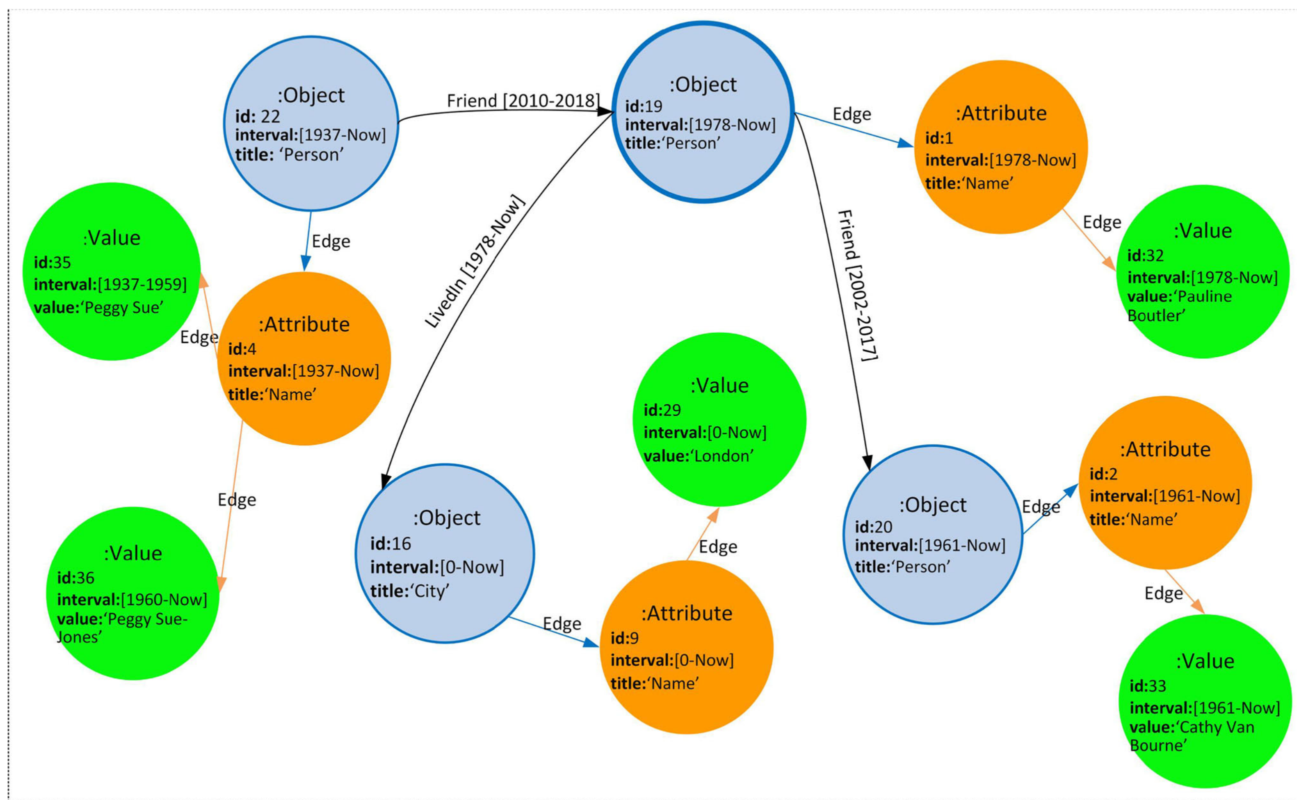
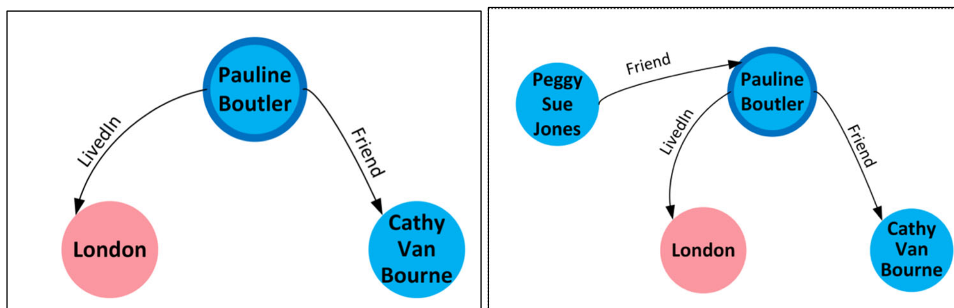


Fig. 7 Social network underlying implementation for Pauline

hovering over or selecting the node. On the right-hand side of the same figure, a Person node can be seen (actually an Object node with title = Person). 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.

5.2 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, a query may include temporal conditions with a granularity different than the one of the database objects, and T-GQL deals with this in a transparent way. For example, to compare the year 2020 against the day 10-20-2020, both dates must have the same granularity. Thus, a conversion is performed at the database level (we do not give here details on how the conversion is done). However, the visual interface must deal with this issue when handling time visualization constraints. 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 starting and ending



(a) Abstraction for Pauline in 2005. (b) Abstraction for Pauline in 2015.

Fig. 8 Abstraction for Pauline

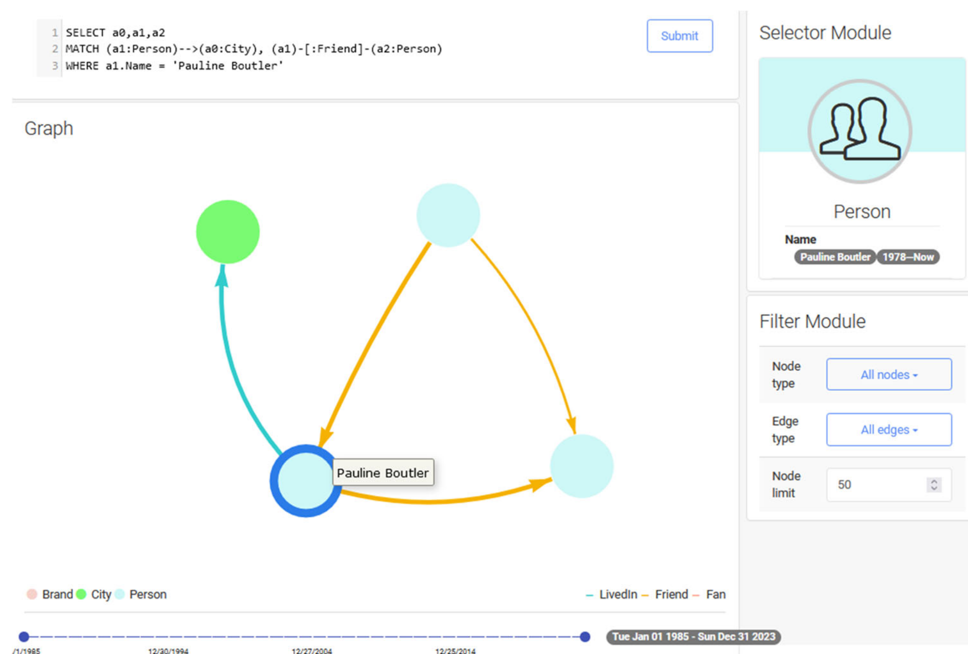


Fig. 9 The graph in Fig. 7 as seen in the visual interface

points of the intervals of the properties. In short, there are two intervals, which 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.

As an example of the above, consider again the graph of Fig. 7 and a query asking for the friends of Pauline during 2005. Two comparisons are needed here, since Pauline has two friends: Peggy Sue and Cathy. The query filter in this case is the interval [2005-01-01 - 2005-12-31], thus, the first comparison involves the filter and the validity interval of her friendship with Peggy Sue, namely, [2010-01-01 - 2018-12-31]. In this case, the intervals do not overlap. The second comparison is between the filtering interval and the friendship interval with Cathy, that is, [2002-01-01 - 2017-12-31]. The intersection between those intervals is [2005-01-01 - 2005-12-31]. Figure 8a shows the resulting graph.

5.3 Rebuilding the Temporal Graph

T-GQL queries are submitted to the engine from the so-called query box shown in Section 4. The engine returns the result in a JSON structure. Thus, in order to display the temporal graph on the main panel, it must be rebuilt from the query result. The graphic 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. In the JSON structure mentioned above, the identifiers of the returned nodes are stored in the id property. Since the information returned by the TGDB is not enough to display the graph (i.e., information about edges is not included), the Neo4j database must be accessed through a Cypher query to obtain the nodes that participate in the T-GQL query result, together with all their relationships, taking into account also the filters applied by the user. In summary, once the query is submitted by the user, the following steps are executed:

Step 1 Rebuild the user T-GQL query string to:

(a) remove the properties in the SELECT clause
(b) get the selected nodes requested in the WHERE clause

Step 2 Run the modified T-GQL query obtained in the previous step.
Step 3 Parse the resulting JSON structure to obtain the list of node ids involved in the query result ([nodeIds]).
Step 4 Execute the Cypher 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]
```

The query above returns a list with the type of edge between every pair of nodes involved in the query along with its interval.

Step 5 If the query returns paths, create the set *restricted edges* containing all edges in the result, along with their direction.

Step 1 is basically a query preprocessing task, explained next. We need to replace the attributes in the SELECT clause with node variables, so the query can return, in the JSON document, all the information needed to rebuild the graph. For example, in Step 1 (a), the following query:

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

will be transformed into:

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

Note that the original query would just return a list of city names, and thus the identifiers of those nodes will be the ones of the *Value* nodes rather than the identifiers of the Object nodes. To obtain the edges involved in the query, the ids of the corresponding *Object* nodes are needed. By converting c.Name to c, the engine will return a list of the ids of the Object nodes. With such list, the edges can be retrieved. The attributes that the user needs are not relevant in this case, because the displayed graph shows all the attributes in the selector module. The process above is carried out by means of a simple string search by finding the start of the SELECT clause and the start of the MATCH clause, and then obtaining the sub-string in-between these indices (a string containing a number of variables or variables mentioning an attribute after a dot, separated by commas.) 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 removed.

For Step 1 (b), the WHERE clause is parsed, to find all the items requested by the query, in order 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 expression is used to obtain a list of terms of the form *Word* = *Word* (considering blanks, enter, tabs, etc), after the WHERE clause. The second one receives every expression obtained with the previous regular expression and will extract the requested value. Thus, expressions of the form *variable.attribute* = *constant* are retrieved. These values are stored in a array that is then used to highlight the desired nodes. For instance, consider the following WHERE clause (see Example 1 below):

```
p2.Name = 'Pauline Boutler' and
cPath((p1)-[:Friend*2]->(p2))
```

Applying the first regular expression, the term *p2.Name = 'Pauline Boutler'* will be obtained, since the expression after the “and” does not involve the “=” symbol. Then, the second expression will store the attribute “Name” and its value “Pauline Boutler”.

After the preprocessing above, we are ready to continue with the next steps. To execute Step 2, the application hooks up to the T-GQL backend through an endpoint, using it as service provider. Similarly, to execute the Cypher query in Step 4, the application access Neo4j through an endpoint. In Step 5, the set *restricted edges* is built from the edges obtained in Step 4, and it is used to prevent inserting (and displaying) the same edge twice with different directions.

The user can narrow the types and number of nodes to be shown when the graph is rebuilt and displayed in the main panel. 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.

Example 1 Consider a query over the database of Fig. 1, that asks for the names of the “friends of the friends” of Pauline Boutler, during a continuous interval, together with information of the latter. The query is expressed in T-GQL as follows.

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

The TGDB service returns the result in JSON format, containing a list of nodes and attributes satisfying the query. The result is shown in Fig. 10. Here, the id property in column **p1** contains the id of the Object node (needed to rebuild the graph), corresponding to Pauline Boutler. However, note that the id property in column **p2.Name** corresponds to the Value Node, since the SELECT clause asks for p1 (a node) and p2.Name (a property). Thus, we apply Step 1 above to modify the query. In this case, the query would be modified as follows (replacing p2.Name with p2):

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

Fig. 10 Result for the original query in Example 1, returned by the database engine, in JSON format

p1	p2.Name
{ "interval": ["1978–Now"], "id": 19, "title": "Person" }	{ "interval": ["1960–Now"], "id": 34, "value": "Peter Green" }

The result obtained from the TGDB is shown in Fig. 11. Now, both ids correspond to the Object nodes, which are necessary to obtain the information on edges not present in this result. Note that the id of the Value node in Fig. 10 was 34, and the id of the Object node in Fig. 11 is 21. Thus, in order to display the graph, the database must be queried again to get this information (Step 3, Step 4). The new query will only involve the nodes in the result of the user query rather than the whole graph. Thus, the identifiers of the nodes in the result are collected, and the Cypher query below is submitted to the Neo4j database:

```
MATCH (n:Object)-[r]->(m:Object)
WHERE n.id in [19,21]
AND m.id in [19,21]
RETURN collect([[n.id,m.id],
type(r), r.interval]
```

The query above returns the information needed to display the graph. The color of different types of nodes and edges will be the ones selected by the user in the previous screen. □

If the original query asks for paths, the process is slightly different. Similarly to the previous case, the returned JSON structure is parsed to obtain the node ids (Step 3) in order to execute Step 4. However, in this case, a color code is associated with the nodes belonging to a certain path. Also, as stated in Step 5, the restricted set of edges is created. The TGDB response returns the nodes in each path in 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, since 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. Further, the color for a node is chosen depending on whether the node is part of a path or not. If it is, we proceed as above. If it is not part of a path, the specific color for that type of node (which was selected in the settings view) is used. Before consolidating this node in the visualization, a verification is made to corroborate that it belongs to the WHERE clause, obtained in Step 1(b), 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.

Fig. 11 Result for the *modified* query in Example 1, returned by the database engine, in JSON format

p1	p2
{ "interval": ["1978–Now"], "id": 19, "title": "Person" }	{ "interval": ["1961–Now"], "id": 21, "title": "Person" }

6 Querying and Visualizing Temporal Graphs

This section is aimed at showing 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 Fig. 1 is submitted. In the second part, a query restricting the social network graph is shown, to convey the idea of how a typical user will operate with the tool presented here, based on the assumption that a user normally focuses on the history of just a part of the graph, or on some kind of summarization of the graph that reduces the usually huge number of nodes that a graph contains. *Again, we remark that operating in this way is not possible in most graph visualization tools*. That is, once the query result is returned, we keep the result and navigate through this result over time, while in typical graph interfaces, the result of a query cannot be used to further navigate the graph (also, recall that not always a query returns a graph, but a table, like in Neo4j).

Figure 12 shows the social network graph in Fig. 1, as seen in the visualizer interface. The query in the top panel retrieves all Person, City, and Brand nodes in the graph:

```
SELECT a0, a1, a2
MATCH (a0:Person), (a1:City), (a2:Brand)
```

The result is displayed in the main panel. The cursor hovers, in this figure, over the Friend relationship (we can see the box with the type and interval of the relationship). It can also be seen on the panel on the right-hand side, the Person node corresponding to Peggy Sue-Jones, along with the history of her name (as a single and married person). This is because, the user clicked over the node corresponding to Peggy, which appears with a thick border in the main panel. We can also see that there are three edges (in blue) from Peggy's node to the cities where she lived: Brussels, Antwerp, and Hasselt (as shown in Fig. 1).

Now, we move the slider in a way such that the graph displayed on the screen is restricted to the interval Dec.2004-Dec.2005. The result is shown in Fig. 13). The Friend relationship in the period 2010-2018 (which was displayed in Fig. 12 with the box over it) is no longer present. Also, since Peggy lived in Brussels between 1979 and 1989, this edge is no longer shown, neither is shown the edge from Peggy to Hasselt, because she started living in the latter in 2020. Finally, hovering over the LivedIn relation (for Peggy) and clicking into the associated city node, we can see, on the right-hand side of the panel, the information about the city of Antwerp (which becomes the node marked with a thick blue line in the graph).

In the next query, the user asks for the friends of Peggy and the cities where they lived, when Peggy was living in Antwerp (that is, between 1990 and 2019). Thus, she writes:

```
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'
```

We can see that the user is only interested in the Person nodes corresponding to Peggy and her friends, and the City nodes corresponding to the cities where Peggy's friends lived. Since the query does not ask for the continuous paths, any

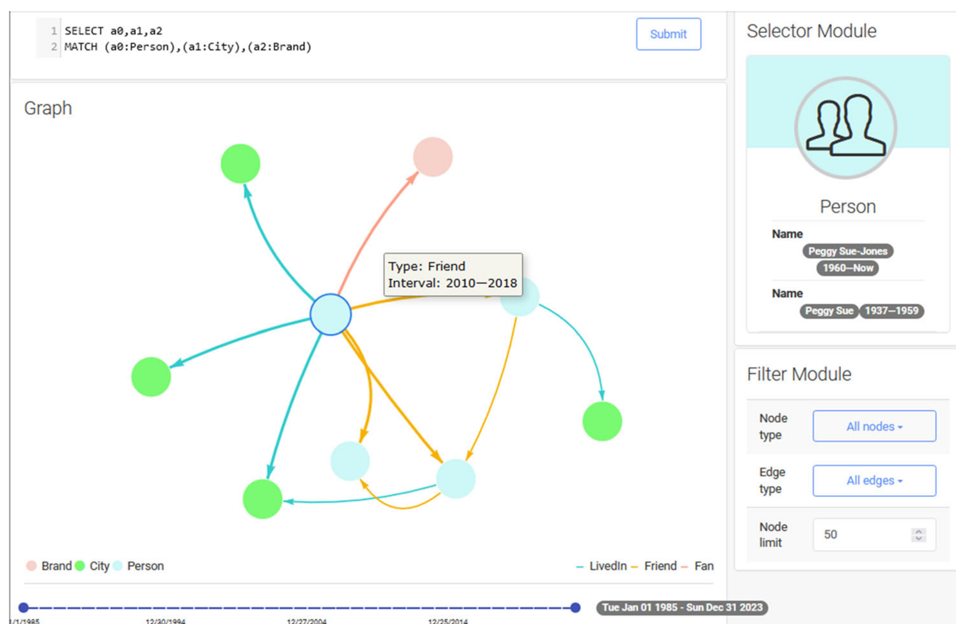


Fig. 12 Social network temporal graph of Fig. 1

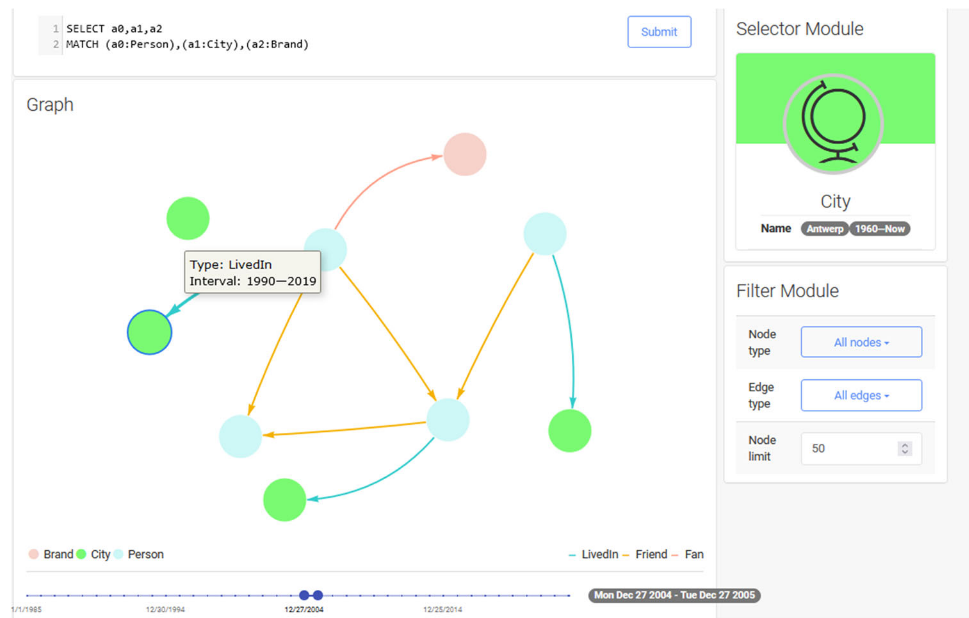


Fig. 13 Social network temporal graph as of the interval Dec.2004-Dec.2005

path in the graph will satisfy the query provided it matches the pattern, regardless the time intervals of the Friend relation. Figure 14 depicts the result. Note that now the user poses the cursor over Pauline, one of Peggy's friends. The node corresponding to Peggy is marked in thick black line since it is in the WHERE clause (as explained in Section Step 1 in Section 5), and the user also selected the Person node corresponding to Cathy, another friend of Peggy. This node is marked in blue thick line. Also note that the LivedIn relationship for the cities where Peggy lived, are not indicated in the figure since they are not requested in the query.

7 Experimental Evaluation

We carried out a series of experiments 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 to estimate the overhead introduced by the visualizer. Thus, relevant to this goal is the size of the graph displayed on the screen (or the

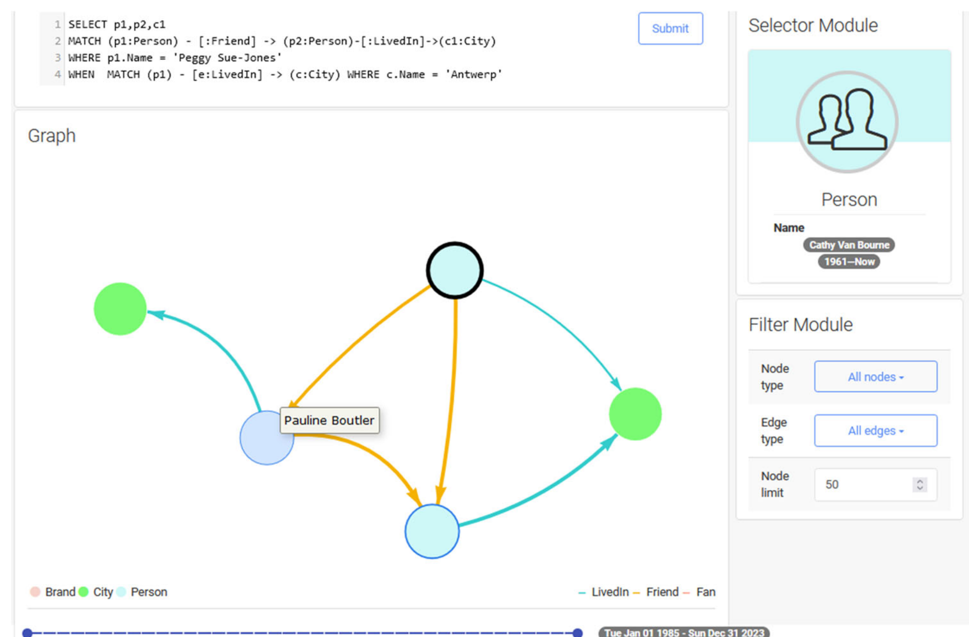


Fig. 14 Peggy's friends when she lived in Antwerp (light blue), and the cities they lived in (green)

number of nodes in a query result) rather than the size of the database, since the latter will impact mainly on the database engine. Also note that, normally, the user will manipulate and navigate across time, small portions of a graph on screen, generally focusing on some relevant objects (e.g., a person, a brand).

Two cases are studied: (a) A social network similar (but larger) to the example discussed throughout the paper, for analyzing continuous paths; (b) A flight schedule network, to address consecutive paths. Queries over these databases are run using the client tool TGDB explained above.

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.

7.1 Social Network Use Case

Three different (synthetic) social network databases were created with an increasing number of nodes and edges. Table 1 shows the characteristics of each database. The database contains three kinds of Object nodes: Persons, Cities, and Brands. There are also two relationships: Friend (between person nodes), and Fan (from a person to a brand). We defined, for each synthetic database, topology parameters that are relevant to the results. For example, the maximum number of friendship relationships (Max friendships) for the Social Network 3 (SN 3) database is twenty-five and a person may be fan of up to ten brands (Max fans). We also guarantee that there will exist in the graph, continuous paths of length five (parameter cPath min Length).

To analyse these databases, we defined five different queries, detailed below. The rationale behind these queries is that they allow studying to what extent the number of nodes in the result impact on the overhead of the visualizer, since, as we will see, they return graphs of different sizes.

Table 1 Parameters used in the creation of social network databases

	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

Query 1 *Retrieve all people, cities and brands in the database at any time.*

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

Query 2 *List all the continuous paths for the Friend relation, with lengths 2 and 3, between Hilton Turner and Jan Dickens.*

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

Query 3 *List all the friends of Hilton Turner and the cities where they lived between 2000 and 2004.*

```
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'
```

Query 4 *Retrieve all the friends of Hilton Turner when she was living in Danaeview (a fictitious place, of course).*

```
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'
```

Query 5 *Find all the friendships of distance 2 for the person with id=250.*

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

Table 2 reports the number of nodes returned for each query and database. This is aligned with our goals, since (as mentioned above), the user will normally want to see a limited number of nodes and edges on the screen, for example, focusing on a person or brand. Thus, we believe that a maximum number of two hundred and fifty nodes in the result (or six hundred, as in the airport use case studied in the next

Table 2 Number of nodes returned by queries Q0 to Q5 for the Social Network databases

Query	Social Network 1	Social Network 2	Social Network 3
Q1	130	N/A	N/A
Q2	3	5	19
Q3	4	14	25
Q4	2	5	6
Q5	3	109	258

Table 3 Difference in response times between TGV and the T-GQL client for the Social Network graph (msecs)

Query	Social Network 1	Social Network 2	Social Network 3
Q1	145	N/A	N/A
Q2	38	43	44
Q3	39	60	65
Q4	42	47	77
Q5	46	121	885

section) seems reasonable as an upper limit to evaluate the overhead of the graphic tool.

We can see that over databases SN 2 and SN 3, Query 1 did not finish within a reasonable time (this is denoted as N/A), because of the high maxFriendships parameter (25 in this case, while in SN 1 this parameter has a value of 5), which makes it hard to compute the paths. We remark that this output is due to the engine and not to the visualizer. We can also see that Queries 1 and 5 (except for SN 1, in the case of Q5) are the ones returning the largest number of nodes. We also remark that response times for Queries 1 and 5 (not detailed here since this is not relevant to our goals) are much higher than for the other queries (the former produce a higher number of nodes in the query result). To speed up query performance, temporal indexing methods are being studied (Kuijpers et al., 2022), however this is beyond the scope of this work, and we have not indexed the graph.

To estimate the overhead introduced by the visualization tool, we measured the difference in the query response times when the queries are run using the T-GQL client and the interface TGV. Table 3 depicts the results. In the first column from the left, we abbreviated the identification of the queries as Q1,...,Q5. It can be seen that, for SN 1, and Queries Q2 to Q5, the difference in response times remains almost constant, around 40 milliseconds. All these queries return less than ten nodes in the result. For Queries Q2 and Q3 in SNs 2 and 3,

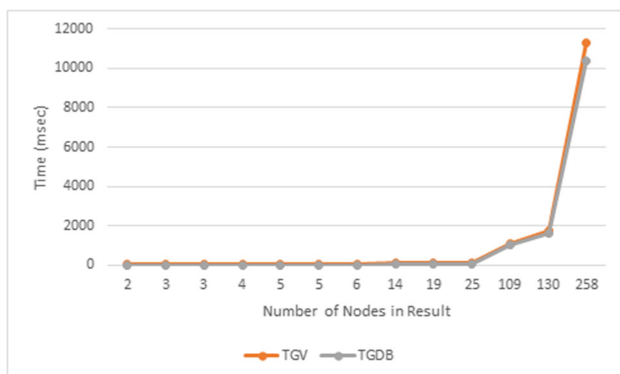


Fig. 15 Response times depending on the number of nodes for the Social Network graph (msecs)

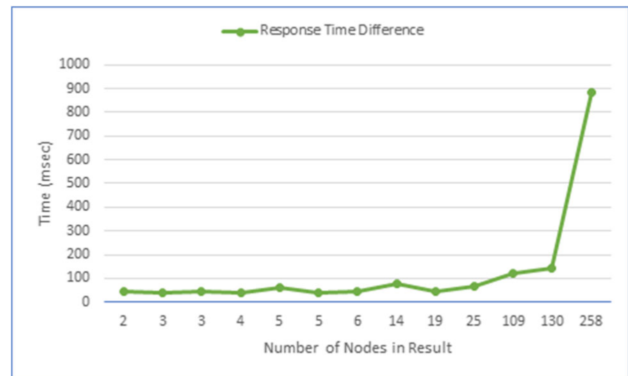


Fig. 16 Difference between the average TGV and TGDB response times depending on the number of nodes for the Social Network graph (msecs)

the difference stays around 60 milliseconds, a little higher than in the previous case, because the number of nodes in the result is slightly above 10. Finally, for Query Q1 in SN 1, and Q5 in SNs 2 and 3, the difference is above 100 msecs, which is consistent with the number of nodes in the result, which is above one hundred. 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 TGDB. The only partial exception to this rule is Query 4, for SN3, with a low number of nodes in the result and a difference of around 80 msecs (probably the size of the database plays a part here). Nevertheless, in all cases, the difference remains lower than one second.

Figure 15 depicts graphically the query response times with respect to the number of nodes in the result (regardless the queries). We can see that the running times using both tools are similar (note that both curves are displayed together, with a small difference between each other). Figure 16 shows the difference between running times with respect to the number of nodes in the result. In both figures we can see that, below 130 nodes, the lines remain almost horizontal, strongly increasing after this value. Our conjecture is that the reason is two-fold: on the one hand, due to the size of the result, the query takes more time to run in nominal terms (as Fig. 15 shows), which also increases the absolute value of the difference. On the other hand, as the size of the result increases, the process described in Section 5 (which involves re-querying

Table 4 Parameters used in the creation of each database

	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

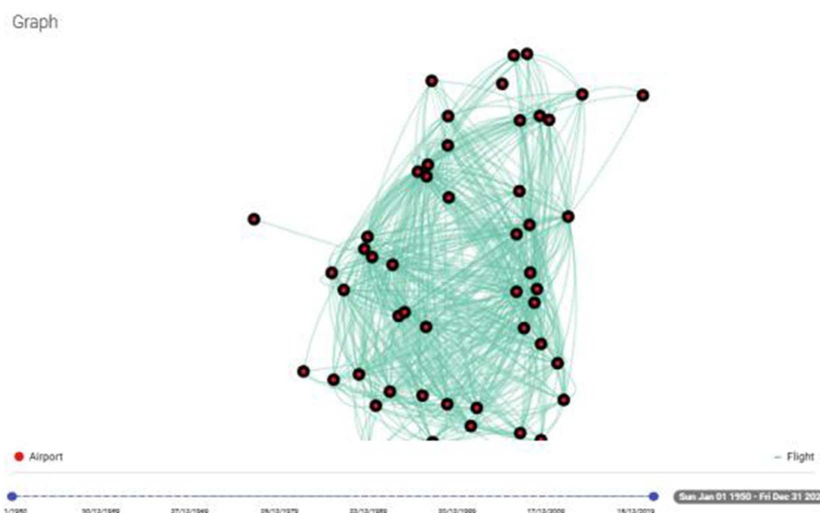


Fig. 17 Airports temporal graph

the database with the ids of the nodes in the result) also takes more time. Further, although we can see a peak when the result size is above 130 nodes, note that the difference still remains below one second (see Fig. 16), which means that the cost of running the query is led by the database engine and not by the visualizer.

7.2 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 4 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 17 shows the complete graph loaded on the visualizer. The queries are listed next.

Query 6 *List all the airports and cities in the database.*

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

Query 7 *Compute the fastest paths between the airports located in the cities of Port Berniecebury and West Hipolitohaven.*

```
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'
```

Query 8 *Compute the latest departure path since 17:04 in 2020-12-11 between Port Berniecebury and West Hipolitohaven.*

```
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'
```

Query 9 *List all the airports and cities that have direct flights from Port Berniecebury.*

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

Table 5 Number of nodes returned by queries Q6 to Q10 for the Airport databases

Query	Airport 1	Airport 2	Airport 3
Q6	20	200	600
Q7	3	2	2
Q8	4	4	2
Q9	3	8	6
Q10	11	45	36

Table 6 Difference in response times between the interface and the T-GQL server for the Airports graph (msecs)

Query	Airport 1	Airport 2	Airport 3
Q6	22	66	130
Q7	18	25	17
Q8	22	14	14
Q9	20	16	14
Q10	21	18	14

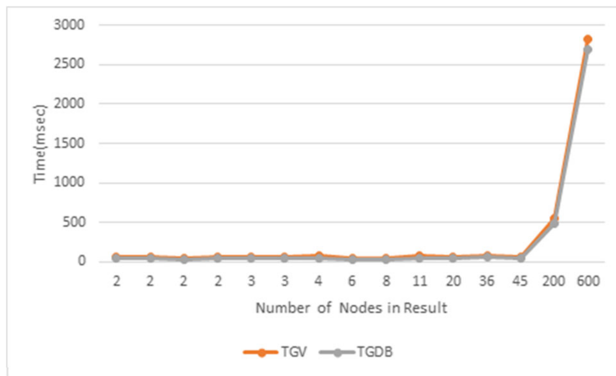


Fig. 18 Response times depending on the number of nodes, for the Airports graph (msecs)

Query 10 *List all the airports and cities at a 4-scale distance from Port Berniecebury. Of course of the city names are fictitious.*

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

Table 5 shows the number of nodes returned for queries Q6 to Q10 and the three Airport databases. Note that the largest results are obtained for Q6 and (in less degree) Q10.

Table 6 shows the results of this second set of experiments, that is, for queries Q6 through Q10. Figure 18 depicts graphically the execution times with respect to the number of nodes in the result (regardless the queries), using the visualizer and the T-GQL client tool. Figure 19 shows the difference between running times with respect to the number of nodes in the result. As in the social network case, with less than 10 nodes in the query result, the response times are quite constant, around 20 milliseconds 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 Q6, 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 Fig. 19. 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.

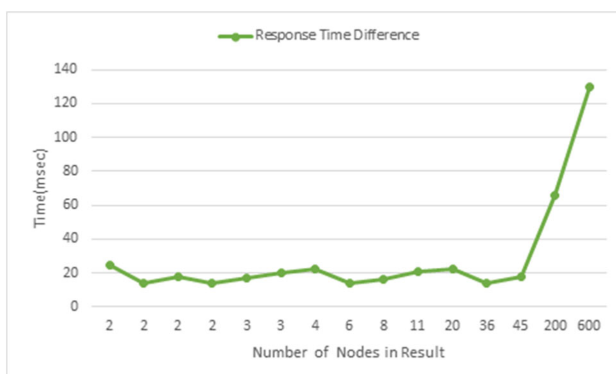


Fig. 19 Difference between the average TVG and TGDB response times depending on the number of nodes, for the Airports graph (msecs)

8 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 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. Experimental results queryPlease check and confirm if the presentation of backfundinfo and conflicts of interest is presented correctly. showed that the overhead introduced by the visualization tool is not relevant, and most of the times negligible.

Funding Alejandro Vaisman and Valeria Soliani were partially supported by Project PICT 2017-1054, from the Argentinian Scientific Agency. Valeria Soliani is also partially supported by the *Bijzonder Onderzoeksfonds* (BOF) from UHasselt with reference BOF22BL02

Declarations

Conflicts of interest 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

Angles, R. (2012). A Comparison of Current Graph Database Models. In: Proceedings of ICDE Workshops, (pp. 171–177). Arlington, VA, USA

Angles, R. (2018).The property graph database model. In: Proceedings of the 12th Alberto Mendelzon International Workshop on Foun-

dations of Data Management, Cali, Colombia, May 21-25, 2018, CEUR Workshop Proceedings, vol. 2100. CEUR-WS.org

Bar, M., & Neta, M. (2006). Humans prefer curved visual objects. *Psychological Science, 17*(8), 645–648. https://doi.org/10.1111/j.1467-9280.2006.01759.x

Bassili, J. N. (1978). Facial motion in the perception of faces and of emotional expression. *Journal of Experimental Psychology: Human Perception and Performance, 4*(3), 373–379. https://doi.org/10.1037/0096-1523.4.3.373

Bertin, J. (1983). Semiology of Graphics. University of Wisconsin Press

Bostock, M. (2017) Force-directed graphs. https://observablehq.com/@d3/force-directed-graph?collection=@d3/d3-force

Brewer, C. A. (2004). 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

Byun, J., Woo, S., & Kim, D. (2020). Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Trans. Knowl. Data Eng., 32*(3), 424–437. https://doi.org/10.1109/TKDE.2019.2891565

Card, S. K., Mackinlay, J. D., & Shneiderman, B. (Eds.). (1999). *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Clifford, J., Tansel, A. (1985). On an algebra for historical relational databases: Two views. In: Proc. of the 1985 ACM SIGMOD International Conference on Management of Data, (pp. 247–265). ACM Press, Austin, TX, USA

Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., & Vaisman, A. (2021). A model and query language for temporal graph databases. *The VLDB Journal*. https://doi.org/10.1007/s00778-021-00675-4

Dignös, A., Böhlen, M., Gamper, J., Jensen, C. (2016). Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries. ACM Transactions on Database Systems, 41(4), 26:1–26:46

Gadia, S. (1988). A homogeneous relational model and query languages for temporal databases. *ACM Transactions on Database Systems, 13*(4), 418–448.

Gamper, J., Ceccarello, M., Dignös, A. (2022). What's new in temporal databases? In: Proc. of the 26th European Conference on Advances in Databases and Information Systems, ADBIS 2022, Lecture Notes in Computer Science, vol. 13389, (pp. 45–58). Springer, Turin, Italy

Gao, Q., Lee, M., Ling, T. (2021) Temporal keyword search with aggregates and group-by. In: Proc. of the 40th International Conference on Conceptual Modeling, ER 2021, Lecture Notes in Computer Science, vol. 13011, (pp. 160–175). Springer

Grandi, F., Mandreoli, F., Martoglia, R., & Penzo, W. (2022). Unleashing the power of querying streaming data in a temporal database world: A relational algebra approach. *Information Systems, 103*, 101872.

Hartig, O. (2014). Reconciliation of RDF* and property graphs. CoRR arXiv:1409.3288

Huo, W., Tsotras, V.J. (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, (pp. 38:1–38:4)

Intelligence, C. (2021) The regraph toolkit. https://cambridge-intelligence.com/regraph/

Jensen, C., Soo, M., & Snodgrass, R. (1994). Unifying temporal data models via a conceptual model. *Information Systems, 19*(7), 513–547.

Jonathan Robie, T.R. (1998) Rec-dom-level-1. https://www.w3.org/TR/REC-DOM-Level-1/introduction.html

Kreitzberg, C. (2017) The intuitive interface. Princeton University, https://ux.princeton.edu/learn-ux/blog/intuitive-interface. Accessed 15 May 2021

Kuijpers, B., Ribas, I., Soliani, V., Vaisman, A.A. (2022) Indexing continuous paths in temporal graphs. In: New Trends in Database and Information Systems - ADBIS 2022 Short Papers, Doctoral Consortium and Workshops: DOING, K-GALS, MADEISD, MegaData, SWODCH, Turin, Italy, September 5-8, 2022, Proceedings, Communications in Computer and Information Science, vol. 1652, (pp. 232–242). Springer

Kulkarni, K., & Michels, J. E. (2012). Temporal features in SQL:2011. *SIGMOD Record, 41*(3), 34–43.

Lima, M. (2009) Information visualization manifesto. Visual Complexity. http://www.visualcomplexity.com/vc/blog/?p=644

Lima, M. (2017). *The Book of Circles: Visualizing Spheres of Knowledge*. NY, NY: Princeton Architectural Press.

Lu, W., Zhao, Z., Wang, X., Li, H., Zhang, Z., Shui, Z., Ye, S., Pan, A., & Du, X. (2019). A lightweight and efficient temporal database management system in TDSQL. *Proc. of the VLDB Endowment, 12*(12), 2035–2046.

Patel, P., Mistry, S. (2016). A guide to material design, a modern software design language. Open Source for You, (pp. 64–66). https://www.opensourceforu.com/2016/05/a-guide-to-material-design-a-modern-software-design-language/

Rizzolo, F., & Vaisman, A. (2008). Temporal XML: Modeling, indexing, and query processing. *VLDB Journal, 1179–1212*(5), 39–65.

Robinson, I., Webber, J., Eifrém, E. (2013). Graph Databases. O'Reilly Media

Semertzidis, K., & Pitoura, E. (2019). Top-k durable graph pattern queries on temporal graphs. *IEEE Trans. Knowl. Data Eng., 31*(1), 181–194.

Snodgrass, R. (ed.) (1995). The TSQL2 Temporal Query Language. Kluwer Academic

Snodgrass, R. (2000). *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann.

Tansel, A., Clifford, J., Gadia (eds.), S. (1993) Temporal Databases: Theory, Design and Implementation. Benjamin/Cummings

Toman, D. (1996) Point vs. interval-based query languages for temporal databases. In: Proc. of ACM PODS, (pp. 58–67). ACM Press, Montreal, Canada

Toman, D. (1997). Point-based temporal extension of temporal SQL. In: Proc. of the 5th International Conference on Deductive and Object-Oriented Databases, DOOD'97, Lecture Notes in Computer Science, vol. 1341, (pp. 103–121). Springer, Montreux, Switzerland

Vartanian, O., Navarrete, G., Chatterjee, A., Fich, L. B., Leder, H., Modroño, C., Nadal, M., Rostrup, N., & Skov, M. (2013). Impact of contour on aesthetic judgments and approach-avoidance decisions in architecture. *Proceedings of the National Academy of Sciences, 110*, 10446–10453. https://doi.org/10.1073/pnas.1301227110

Zimányi, E. (2006). Temporal aggregates and temporal universal quantifiers in standard SQL. *SIGMOD Record, 32*(2), 16–21.

Zimányi, E., Sakr, M., Lesuisse, A. (2020). MobilityDB: A mobility database based on PostgreSQL and PostGIS. ACM Transactions on Database Systems, 45(4), 19:1–19:42

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Diego Orlando received a BA in Computer Engineeringfrom the Instituto Tecnológico de Buenos Aires (ITBA), Argentina, in 2021. He is currently a Software Engineer at Accendo, in Buenos Aires, Argentina.

Joaquín Ormachea received a BA in Computer Engineering from the Instituto Tecnológico de Buenos Aires (ITBA), Argentina, in 2020. He is currently a Data Engineer at Salesforce in Buenos Aires, Argentina.

Valeria Soliani received a BA degree in Computer Science from the University of Buenos Aires (UBA) and a Data Science Especialist degree from the the Instituto Tecnológico de Buenos Aires (ITBA). Se is a Ph.D. student at ITBA and at Hasselt University. Her field of study is temporal graph databases, and has published papers in this field in international journals and conference proceedings.

Alejandro Ariel Vaisman received a BA degree in Civil Engineering, a BA in Computer Science, and a PhD in Computer Science from the University of Buenos Aires (UBA), under the supervision of Prof. Alberto Mendelzon, from the University of Toronto, Canada. He was post-doctoral researcher and Lecturer at the University of Toronto. He was Associate Professor at UBA between 1994 and 2013, Vice-Head of the Computer Science Department at UBA, and chair of the Masters Program in Data Mining. He was a visiting researcher at the University of Toronto, Universidad Politécnica de Madrid,University of Hasselt, Universidad de Chile and Université Libre de Bruxelles. He is currently full professor at the InstititNuto Tecnológico de Buenos Aires (ITBA), where he is also Director of the Masters Program in Data Science. His research interests are in the field of databases, particularly in Business Intelligence, OLAP and Data Warehousing, the Semantic Web and Geographic Information Systems. He has authored and co-authored over 100 scientific papers presented at major database conferences and journals, and co-authored the book “Data Warehouse Systems: Design and Implementation” (2nd. ed. published in 2022).