A model and query language for temporal graph databases

Peer-reviewed author version

Debrouvier, Ariel; Parodi, Eliseo; Perazzo, Matías; SOLIANI, Valeria & VAISMAN, Alejandro (2021) A model and query language for temporal graph databases. In: VLDB JOURNAL, 30(5), p. 825-858.

DOI: 10.1007/s00778-021-00675-4
Handle: http://hdl.handle.net/1942/34583

VLDB Journal manuscript No.
(will be inserted by the editor)

A Model and Query Language for Temporal Graph Databases

Ariel Debrouvier · Eliseo Parodi · Matías Perazzo · Valeria Soliani · Alejandro Vaisman

Received: date / Accepted: date

Abstract Graph databases are becoming increasingly popular for modeling different kinds of networks for data analysis. They are built over the property graph data model, where nodes and edges are annotated with property-value pairs. Most existing work in the field is based on graphs were the temporal dimension is not considered. However, time is present in most real-world problems. Many different kinds of changes may occur in a graph as the world it represents evolves across time. For instance, edges, nodes, and properties can be added and/or deleted, and property values can be updated. This paper addresses the problem of modeling, storing, and querying temporal property graphs, allowing keeping the history of a graph database.

This paper introduces 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, a high-level graph query language, denoted T-GQL, is presented, together with a collection of algorithms for computing different kinds of temporal paths in a graph, capturing different temporal path semantics. T-GQL can express queries like *"Give me the friends of the friends of Mary, who lived in Brussels at the same time than her, and also give me the periods when this happened"*. As a proof-of-concept, a Neo4j-based implementation of the above is also presented, and a client-side interface allows submitting queries in T-GQL to a Neo4j server. Finally, experiments were carried out over synthetic and real-world data sets, with a two-fold goal: on the one hand, to show the plausibility of the approach; on the other hand, to analyze the factors that affect performance, like the length of the paths mentioned in the query, and the size of the graph.

Keywords Temporal Graph Databases · Neo4j · Query Languages · Cypher Query Language · Graph databases

Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, 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: {ndebrouvier,eparodi,maperazzo,vsoliani,avaisman}@itba.edu.ar

1 Introduction and Motivation

Property Graphs [4,28,44] have been increasingly gaining popularity, especially for modeling and analyzing different kinds of networks. The property graph data model underlies most graph databases in the marketplace [2]. Examples of graph databases based on this model are Neo4j[1], Janusgraph[2], and GraphFrames [16]. Typically, the work of researchers and practitioners is based on graphs were the temporal dimension is not considered, called static graphs hereon. However, time is present in most real-world applications, and graphs are not the exception. 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 the most relevant ones. For instance:

- (a) In a phone call network, where each vertex represents a person (or a phone number), and an edge (u, v, t, λ) tells that u calls v at time t, with duration

[1] `http://www.neo4j.com`
[2] `http://janusgraph.org/`

λ, new nodes and edges are added frequently, and also the properties of u or v may change over time.
- (b) In social networks (e.g., Facebook, Twitter), each vertex models a person (or an organization, etc.), and an edge (u, v, t, λ) represents a relationship between two persons u and v (e.g., u follows v, u is a friend of v) at time t which lasts λ (u was a friend of v during an interval whose duration is λ).
- (c) In transportation networks, each vertex represents a location, and an edge (u, v, t, λ) represents a road segment, a street, or a highway segment, from u to v, existing since time t, and whose interval of existence is λ.
- (d) In transportation schedules, each vertex in a graph represents a location, and an edge (u, v, t, λ) is a trip (flight, bus, etc.) from u to v departing at time t, whose duration is λ.

Ignoring the time dimension could lead to incorrect results, or prevent interesting analysis possibilities. For example, in case (b), 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. As another example, a user may be interested in asking for "People who were still being Nutella fans while they were living outside Italy," or "Friends of Mary while she was working at the University of Antwerp." Those are queries that could not be answered without accounting for time. As another kind of problem, note that in case (c) above, the shortest (or fastest) way to reach one city from another one, varies with time, since a segment belonging to the shortest path may have not existed in the past. Thus, for example, a transportation analyst may ask for the "Time saved for going from Buenos Aires to Pinamar after the construction of Highway Number 11." Further, this can be stated as a hypothetical query [7,22], asking for the fastest way to reach a city in case a new highway is built.

Literature in temporal graphs is relatively limited, and basically oriented to address path problems particularly for scenarios like (a) and (d) above. As far as the authors are aware of, problems tackling scenarios like (b) and (c), which require an approach over property graphs along the lines of the temporal databases theory [48] have not been addressed yet, with some partial exceptions discussed in Section 2. Temporal property graph-based data models, query languages, algorithms, and even, a study of the problems that can be solved with this approach, are still open fields of study, and this work tackles them.

1.1 Contributions

This paper studies how temporal databases concepts can be applied to graph databases, in order to be able to model, store, and query temporal graphs, in other words, to keep the history of a graph database. The work presented here is based on the property graph data model. This is not the case of most existing work on the topic (e.g., [52,54]), where only edges are timestamped with the initial time of the relationship they represent, and the duration of such relationship. Further, those works address homogeneous graphs (i.e., graphs were only one kind of relationship exists). In the model presented here, nodes, relationships, and node properties are timestamped with their temporal validity interval, and graphs are heterogeneous, that means, relationships may be of different kinds. These graphs are called Interval-labeled Property Graphs in this paper. This allows richer queries, like *"Give me the friends of the friends of Mary, who lived in Brussels at the same time than her"*. Nevertheless, the model presented in this paper also captures the semantics of the mentioned works. For this, two path semantics are supported: Continuous path semantics, defined along the lines of the work by Rizzolo and Vaisman [43], and Consecutive Path Semantics. Both semantics, and their implementation, are discussed in detail. More concretely, the contributions of this work are:

- A temporal graph data model for property graphs, which allows keeping the history of nodes, edges, and properties.
- A high-level graph query language, denoted T-GQL, based on GQL[24] (standing for Graph Query Language), the standard language for property graph databases being defined by the graph database community at the time of writing this paper.
- A collection of algorithms for computing different kinds of temporal paths in a graph, capturing different temporal path semantics.
- A Neo4j-based implementation of the above, and a client interface for querying Neo4j graphs.
- A collection of experiments over the implementation, over two use cases which capture the two semantics studied in this work: a synthetic data set of a social network, and a real-world data set of flights between airports.

1.2 Paper Organization

This paper is organized as follows. Section 2 reviews related work, in order to put the present work in context. Section 3 introduces the temporal property graph

data model that will be used in the paper, and Section 4 presents and discusses T-GQL, the high-level data language proposed for the model, and Section 5 presents the implementation details. Section 6 reports preliminary experimental results. Section 7 studies how query performance can be enhanced indexing the different kinds of paths defined in the paper. Section 8 concludes the paper.

2 Related Work

There is a large corpus of work in the field of temporal relational databases, over which the present work builds [48,19]. TSQL2 [47] is the temporal extension to SQL, proposed to the international standardization committees, and some of its features are included in the SQL:2011 standard [38]. Further, the temporal relational model has inspired temporal extensions for different data models [13], like XML [1,14,43]. Given that this literature is well known, this section addresses work related with graph models, starting from traditional (non-temporal) property graphs, and then moving on to the few existing work on temporal graphs. These existing proposals are compared against the work presented here.

2.1 Graph database models

There is an extensive bibliography on graph database models, comprehensively studied in [2,5]. The interested reader is referred to these works for details. Multiple native graph indexing methods and query languages (e.g., GraphQL [31]) were developed to efficiently answer graph-oriented queries. In real-world practice, two graph database models are used:

(a) Models based on RDF,[3] oriented to the Semantic Web.
(b) Models based on Property Graphs.

Models of type (a) represent data as sets of triples where each triple consists of three elements that are referred to as the subject, the predicate, and the object of the triple. These triples allow describing arbitrary objects in terms of their attributes and their relationships to other objects. Informally, a collection of RDF triples is an RDF graph. Although the models in (a) have a general scope, RDF graphs aim at representing metadata on the Web. Therefore, an important feature of RDF-base graph models is that they follow a standard, which is not yet the case for the other graph databases.

[3] `https://www.w3.org/RDF/`

Temporal extensions for RDF have been proposed. Gutiérrez et al. introduced time in RDF [25,26] by means of timestamping RDF triples with their validity intervals, using the notion of reification. Over this work, extensions to SPARQL, the RDF's standard query language, were proposed [49,23].

In the *property graph* data model [3,4], nodes and edges are labeled with a collection of (attribute, value)-pairs. Property graphs extend traditional graph models, and are the usual choice in modern graph databases used in real-world practice. Hartig [28,29] proposes a formal way of reconciling both models, through a collection of well-defined transformations between property graphs and RDF graphs. He shows that property graphs could, in the end, be queried using SPARQL. This is also studied in [6,51].

Since the problem studied in this paper is based on the property graph model, the review presented next only addresses this graph data model.

2.2 Data models for temporal graphs

Data models in the temporal graphs literature can be classified in three groups:

- (a) Duration-labeled temporal graphs (DLTG)
- (b) Interval-labeled temporal graphs (ILTG)
- (c) Snapshot-based temporal graphs (SBTG)

Graphs of type (a) are typically proposed for the phone calls and travel scheduling problems described above. Graphs of type (b) are more appropriate than the former ones, to capture the history of the relationships in social networks. Graphs of type (c) are based on the notion of snapshot temporal databases, where a temporal database is seen either as a sequence of snapshots, or a sequence composed of an initial database and a sequence of incremental updates. These models are discussed next.

2.2.1 Duration-labeled temporal graphs

These kinds of graphs are studied by Wu et al. [52]. In these graphs, a node is represented as a string (i.e., nodes are not annotated with properties), and the edges are labeled with a value representing the duration of the relationship between two nodes. Based on this work, the same authors have elaborated different proposals [32, 53–56]. All of them address the previously mentioned kinds of graphs. Definition 1 formally explains the above description.

Definition 1 (Duration-labeled graphs (cf. [52])) Let $G_d = (V, E)$ be a temporal graph, where V is the set of vertices, and E is the set of edges in G.

- Each edge $e = (u, v, t, \lambda) \in E$ is a temporal edge representing a relationship from a vertex u to another vertex v starting at time t, with a duration λ. For any two temporal edges (u, v, t_1, λ_1) and (u, v, t_2, λ_2), $t1 \leq t2$.
- Each node $v \in V$ is active when there is a temporal edge that starts or ends at v.
- $d(u, v)$: the number of temporal edges from u to v in G_d.
- $E(u, v)$: the set of temporal edges from u to v in G, i.e., $E(u, v) = \{(u, v, t1), (u, v, t2), ..., (u, v, td(u, v))\}$.
- $N_{out}(v)$ or $N_{in}(v)$: the set of out-neighbours or in-neighbours of v in G_d, i.e., $N_{out}(v) = \{u : (v, u, t) \in E\}$ *and* $N_{in}(v) = \{u : (u, v, t) \in E\}$.
- $d_{out}(v)$ or $d_{in}(v)$: the temporal out-degree or in-degree of $v \in G_d$, $d_{out}(v) = \sum_{u \in N_{out}(v)} d(v, u)$ and $d_{in}(v) = \sum_{u \in N_{in}(v)} d(u, v)$.

Graphs defined in this way are called *Duration Labeled*. The left-hand side of Figure 1 shows an example, where, for simplicity, $\lambda = 1$. □

As mentioned, the main use of this kind of temporal graphs is, for example, for scheduling problems, where usually some sort of shortest path must be computed. Therefore, the works around this model propose 'temporal' variants of the well-known Dijkstra's algorithm [17]. In [52] (and the sequels of this work, referred above), the authors also define four different forms of 'shortest' paths. These are called here *minimum temporal paths*, and account for different measures: (1) *Earliest-arrival path*, defined as a path that results in the earliest arrival time starting from a source x to a target y; (2) *Latest-departure path*, defined as a path that gives the latest departure time starting from x in order to reach y at a given time; (3) *Fastest path*, defined as the path that goes from x to y in the minimum elapsed time ; and (4) *Shortest path*, defined as the path that is shortest from x to y in terms of overall traversal time along the edges.

2.2.2 Interval-labeled temporal graphs

Two main approaches exist in the temporal databases literature [48], for keeping the history of a database: tuple or attribute-timestamping, where a temporal label is defined over the database objects; or database versioning, where different versions of a database are created at different time instants. The latter is described below in this section. The former is discussed next. Definition 2 below, characterizes interval-labeled temporal graphs (ILPG). From this general definition, different constraints can be stated, leading to different models, as the one introduced below in this paper, based on the work by Campos et al [10] (see Section 3), the first approach to apply the ILTG notion to property graphs. Transaction time is considered in the remainder, that is, the time where the information is stored in the database, opposite to valid time, which reflects the time where the data is valid in the real world. This will make the presentation simpler, particularly when discussing updates (Section 3.4) However, as it is discussed later, a limited form of retroactive updates is also allowed, which means that the model supports both kinds of times up to a certain extent.

Definition 2 (Interval-labeled temporal graphs) Let $G_d = (V, E)$ be a temporal graph, where V is the set of vertices, and E is the set of edges in G. A *Duration Labeled Temporal Graph.* is a temporal graph where each edge $e = (u, v, I) \in E$ is a temporal edge representing a relationship from a vertex u to another vertex v, valid during a time interval $I = [t_s, t_e]$. □

The right-hand side of Figure 1 shows a graph equivalent to the one on the left of such figure, but where edges are labeled with their validity interval instead of a timestamp representing a duration. In the ILTG on the right-hand side of Figure 1, for example, the edge between nodes b and g is labeled with the interval $[3, 4]$. This is due to the fact that the same edge, on the left-hand side of the same figure is labeled 3, representing the initial time of the edge, with a duration of 1. That means, if the graph represents a bus schedule, the bus leaves from b at time instant 3, and the trip between b and g takes one time unit.

Example 1 The path traversal times in Section 2.2.1 are also valid in this representation. Consider for example, the computation of the earliest arrival time from node a to every node in the graph, in the interval $[1, 4]$. The algorithm proposed in [52] gives as a result $eat(b) = 2$, $eat(g) = 4$, $eat(h) = 4$, and $eat(f) = 4$. Obviously, this can also be computed with the interval-labeled graph. It it easy to see that, for instance, $eat(g) = 4$, with path $\langle (a, b, [1, 2]), (b, g, [3, 4]) \rangle$, since $\langle (a, b, [2, 3]), (b, g, [3, 4]) \rangle$ cannot be used since the arrival time at b is equal to the departure time from b to g. □

The discussion above gives the intuition that ILTGs and DLTGs are equivalent, in the sense that both allow representing the same information using different encodings for time (the proof is outside the scope of this paper). ILTGs appear to be, at first sight, more appropriate than DLTGs to support classic temporal queries, for example, the ones asking for the history of relationships in a social network. At the same time, travel schedules and mobility data can also be modeled in this way, as Example 1 shows. Moreover, current graph databases are based on the property graph

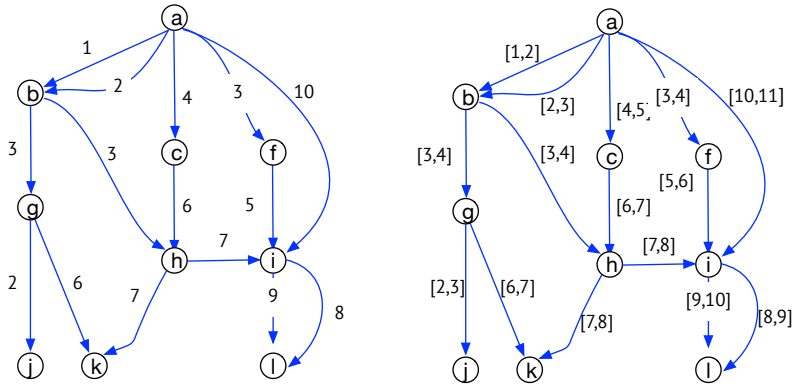

Fig. 1 Left: A duration-labeled temporal graph (cf. [52]); Right: An Interval-labeled temporal graph for the graph on the left.

data model, which are not supported in the work by Wu et al. Therefore, the data model in the present paper along with its the accompanying query language, are based on interval-labeled property graphs.

2.2.3 Temporal graphs as a sequence of snapshots

The work by Semertzidis and Pitoura [46] aims at finding the most persistent matches of an input pattern in the evolution of graph networks. The authors assume that the history of a node-labeled graph is given in the form of graph snapshots corresponding to the state of the graph at different time instants. Given a query graph pattern P, the work addresses the problem of efficiently finding those matches of P in the graph history that persist over time, that is, those matches that exist for the longest time. These queries are called *graph pattern queries*. Locating durable matches in the evolution of large graphs has many applications, like for example, long-term collaborations between researchers, durable relationships in social networks, and so on. In [46], a temporal graph is defined as follows, which defines the third category of temporal graphs introduced above.

Definition 3 (Snapshot temporal graph (cf. [46])) A temporal graph $G[t_i, t_j]$ in a time interval $[t_i, t_j]$, is a sequence $\{G_{t_i}, G_{t_{i+1}}, ..., G_{t_j}\}$ of graph snapshots.

Huo and Tsotras [33] study the problem of efficiently computing shortest-paths on evolving social graphs. The authors define a temporal graph as an initial snapshot, followed by updates. The traditional Dijkstra's algorithm [17] is extended, to compute shortest-path distance(s) for a time-point or a time-interval, within a social graph's evolving history. Temporal queries are thus issued on certain historical graph snapshot(s). For example, temporal shortest-path queries in a social network can discover how close two given users were in the past and how their closeness evolved over time. Finally, several different kinds of path queries are defined. For example, a *time point shortest path query* returns the shortest-path p from a source node v_s to a target node v_t, such that both are temporally valid at query time t_q (all edges in p are valid at query time t_q).

2.2.4 Other work on temporal graphs

Catutto et al. [11,12] present a temporal data model where temporal data are organized in so-called frames, namely the finest unit of temporal aggregation. A frame is associated with a time interval and allows retrieving the status of the social network during such interval. This model does not support changes in the attributes of the nodes. Also, frame nodes may become associated with a large number of edges. Redundant data are also a problem since each frame is connected to all the existing data, so a frequently changing graph would become full of redundant connections.

Khurana and Deshpande [35,36] study methods to efficiently query historical graphs. They focus on the particular problem of querying the state of a network as of a certain point (snapshot) in time. The work is based on versioning. Basically, the current graph and a series of deltas containing the graph variation over time are stored. Among other works related with temporal graphs, Han et al. [27] present an engine for temporal graph mining, and Kostakos [37] shows the use of temporal graphs to represent dynamic events.

Johnson et al. [34] introduce Nepal, standing for Network Path Language, specifically oriented to time-travel path queries over communication networks that can change their state over time. The authors define a temporal inventory, a structure where changes over nodes and edges in the network are recorded. Using a notion similar to the one of continuous paths, a *valid pathway at time t* is defined as a pathway whose nodes

and edges are all valid at time t. In this way, the status of a network at a given time can be obtained. An SQL-like query language is described through examples. For query evaluation, queries are translated to a so-called pathway algebra. The idea is that a pathway is the first-class citizen in this language, and operators are basically conditions over these pathways. However, many issues arising in temporal databases are not addressed (e.g., granularity and complex temporal operations). Further, no implementation is reported.

Lazarevic [40] shows how the versions of a graph can be maintained and queried in Neo4j using Cypher, Neo4j's high-level query language [21]. Although interesting from a practical point of view, the proposal is adhoc rather than an effort to produce a temporal graph database. Path queries are not discussed in this work.

Byun et al. [8,9] address the problem of computing path traversals in large temporal graphs. In [9] the authors introduce ChronoGraph, a system that performs path traversals satisfying temporal constraints on paths. ChronoGraph reconciles point-based and interval-based semantics, in the sense of the notion of telic and atelic temporal data [50]. This is also the case of the model and language presented in this paper (as follows from Sections 2.2.1 and 2.2.2). The paper presents three kinds of temporal path traversal algorithms, implemented on top of ChronoGraph: temporal breadth-first search, temporal depth-first search, and temporal single source shortest path. In these algorithms, temporal paths are traversed considering the temporal labels of the graph's edges. A prototype implementation of ChronoGraph on top of the Tinkerpop framework is also presented. An extension of Gremlin [45] is used as the query language, that is, ChronoGraph's language is imperative (which makes sense, since the system is aimed at path traversal). On the contrary, the model presented here defines its own declarative high-level query language, T-GQL, designed along the lines of traditional temporal databases languages. Also, in [9], each node in the temporal graph is associated with a collection of *static* (property, value) pairs. Conversely, in the model of Section 3 property evolution is supported.

2.2.5 Data Models Comparison

This section discusses the main differences between the model proposed in this paper, and the works commented above.

First, the works in [32,52–56] address unlabelled homogeneous graphs. The same applies to the snapshot-based models discussed in Section 2.2.3. Opposite to the former, the model proposed in the present paper is based on the property graph data model, and supports heterogeneous graphs. Also, computing the temporal paths presented in this paper over the snapshot-based models would be computationally expensive.

Second, in the work by Byun et al. [8,9], temporal properties defined over the nodes are represented as a collection of static (property, value) pairs, while in the model introduced in Section 3, temporal properties are first-class citizens. Further, the temporal query language associated with the former is based on Gremlin, a procedural language appropriate for path traversals, while the present paper introduces T-GQL, a high-level declarative language built along the lines of classic temporal database semantics. This is not a minor difference, since, as follows from the discussion in Section 5.4 below, generalizing Gremlin to address continuous and consecutive temporal paths is not a trivial task.

Third, although the model proposed in [34] supports the notion of continuous path, the work does not dive into many problems arising in temporal databases, and also does not report an implementation or practical results. Moreover, the query language is limited to so-called pathway queries. On the other hand, the model and query language presented in this paper support three different path semantics, namely continuous, pairwise continuous, and consecutive (see Section 3), as well as a wider spectrum of temporal queries.

To close this section, it is worth mentioning that the works discussed in this section do not tackle problems that are typical in a temporal database context, since those works are mainly focused on the problem of computing path traversals. For example, temporal constraints are implicit in the data models supporting the works above, while the temporal data model presented in Section 3 states such constraints explicitly (Definition 6). In addition, the temporal database approach allows the model to support queries that mention time granularities that differ from the ones in the database (Section 4.4). The approach also allows T-GQL to express path queries of different kinds, as well as a rich variety of temporal queries (see Section 4) which other temporal graph data models do not support. For example, queries that ask for paths valid in an interval defined explicitly (using the BETWEEN clause) or determined by a condition (through the WHEN clause), like in the query "*Who were friends of Mary while she was living in Antwerp?*". Further, queries returning a graph as of a certain instant using the SNAPSHOT clause can also be expressed. To the best of the present paper authors' knowledge, no temporal graph data model supports these kinds of queries together with the features presented in this section.

3 A Data Model for Interval-labeled Property Graphs

Property graphs are graphs such that their nodes and edges are labeled with a collection of (property,value) pairs. These properties can evolve over time. Therefore, in order to keep the history of the graph, the data model must not only account for the changes in the relationships and the nodes, but also for the changes in the properties. A first approach for this was presented by Campos et al. [10]. Definition 4 builds on that work.

Definition 4 (Temporal property graph) A *temporal property graph* is a structure $G(N_o, N_a, N_v, E)$ where G is the name of the graph, E is a set of edges, and N_o, N_a, and N_v are sets of nodes, denoted *object nodes*, *attribute nodes*, and *value nodes*, respectively. 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. For any node n, the elements in its associated pair are referred to as `n.name`, `n.interval`, and (for value nodes) `n.value`. In addition, nodes and edges in G satisfy the constraints in Definition 6 below. As usual in temporal databases, a special value Now is used to tell that the node is valid at the current time (see Section 3.4 for more details on this). All nodes also have an identifier, denoted `id`. □

In Definition 4, object nodes represent entities (e.g., Person), edges represent relationships between object nodes (e.g., LivesIn, FriendOf), attribute nodes describe entities (e.g., Name); Finally, value nodes represent the value of an attribute (e.g., Mary). To illustrate this more in detail, the first running example that will be used in this paper is presented next.

Example 2 (Data model) The model in Definition 4 is used to represent the social network depicted in Figure 2. There are three kinds of object nodes, namely Person, City, and Brand. There are also three types of 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 temporal semantics of the relationship Fan is similar. For example, there is an edge of type Fan, joining nodes 14 (a Person node) and 70 (a Fan node), indicating that Mary Smith is a Samsung fan since 1982. The attribute node Name represents the name associated with a Person node, and it is also temporal. The actual value of the attribute node is represented as a value node (represented as ellipses in Figure 2), e.g., the node in green with id=34 and value "Mary Smith". Note that this value changes to "Mary Smith-Taylor", showing the temporality of the attribute node Name. Finally, for clarity, if a node is valid throughout the complete history, the temporal labels are omitted. □

Note that edges could also have properties. However, for simplicity, they are assumed to remain constant throughout their lifespan, that is, they cannot change. Although this is of course a limitation of the model, this assumption is reasonable for most cases, and contributes to readability, without keeping out any fundamental issue. Further, it is worth pointing out that this proposal is implemented at this stage over Neo4j, and this database does not allow indexing edge properties.

Before introducing the temporal graph's constraints, some notation is needed. In Definition 6 below, an edge is denoted by $e\{n_a, n_b\}$ where n_a and n_b are nodes connected by the edge e. An attribute node will be represented as $n_a\{n\}$ where n is the object node connected to n_a. A value node is denoted $n_v\{n_a\}$ where n_a is the attribute node connected to n_v. Also, the following definition is needed.

Definition 5 (Lifespan of an edge) Consider a node n, and a collection of k edges outgoing from n, $E_{out_i}, i = 1, \ldots, k$ such that Out_i.`name` is the same for all E_{out_i}. Also, let $E_{in_j}, j = 1, \ldots, m$ be the set of m edges with the same name incoming to node n. The union of the temporal labels of all these edges is called the *lifespan* of n, denoted $l(n)$. □

Definition 6 (Constraints) For the graph in Definition 4, the following constraints hold:

1. $\forall n, n' \in N_o,\ n = n' \vee n.id \neq n'.id$
2. $\forall n, n' \in N_a,\ n = n' \vee n.id \neq n'.id$
3. $\forall n, n' \in N_v,\ n = n' \vee n.id \neq n'.id$
4. $\forall n_v\{n_a\}, n'_v\{n_a\} \in N_v,\ n_v = n'_v \vee n_v.value \neq n'_v.value$
5. $\forall e_i\{n, n'\}, e_j\{n, n'\} \in E \wedge e_i.name = e_j.name, e_i = e_j \vee e_i.name \neq e_j.name$
6. $\forall n \in N_o,\ e\{n, n'\} \in E \Rightarrow n' \in N_o \bigcup N_a$
7. $\forall n \in N_a,\ e\{n, n'\} \in E \Rightarrow n' \in N_o \bigcup N_v$
8. $\forall n \in N_v,\ e\{n, n'\} \in E \Rightarrow n' \in N_v$
9. $\forall n \in N_a (\exists\ n_o \in N_o\ \exists e \in E\ (e(n_o, n) \wedge (\nexists n' \in (N_a \bigcup N_v \bigcup N_o) \wedge e' \in E \wedge e'\{n', n\})))$
10. $\forall n \in N_v\ (e\{n', n\} \wedge n' \in N_a) \Rightarrow \nexists n'' \in (N_a \bigcup N_v \bigcup N_o)\ (e''\{n'', n\} \in E \vee e''\{n, n''\} \in E)$
11. $\forall n_e\{n, n'\} \in N_e, n_e.interval \subset n.interval \cap n'.interval$

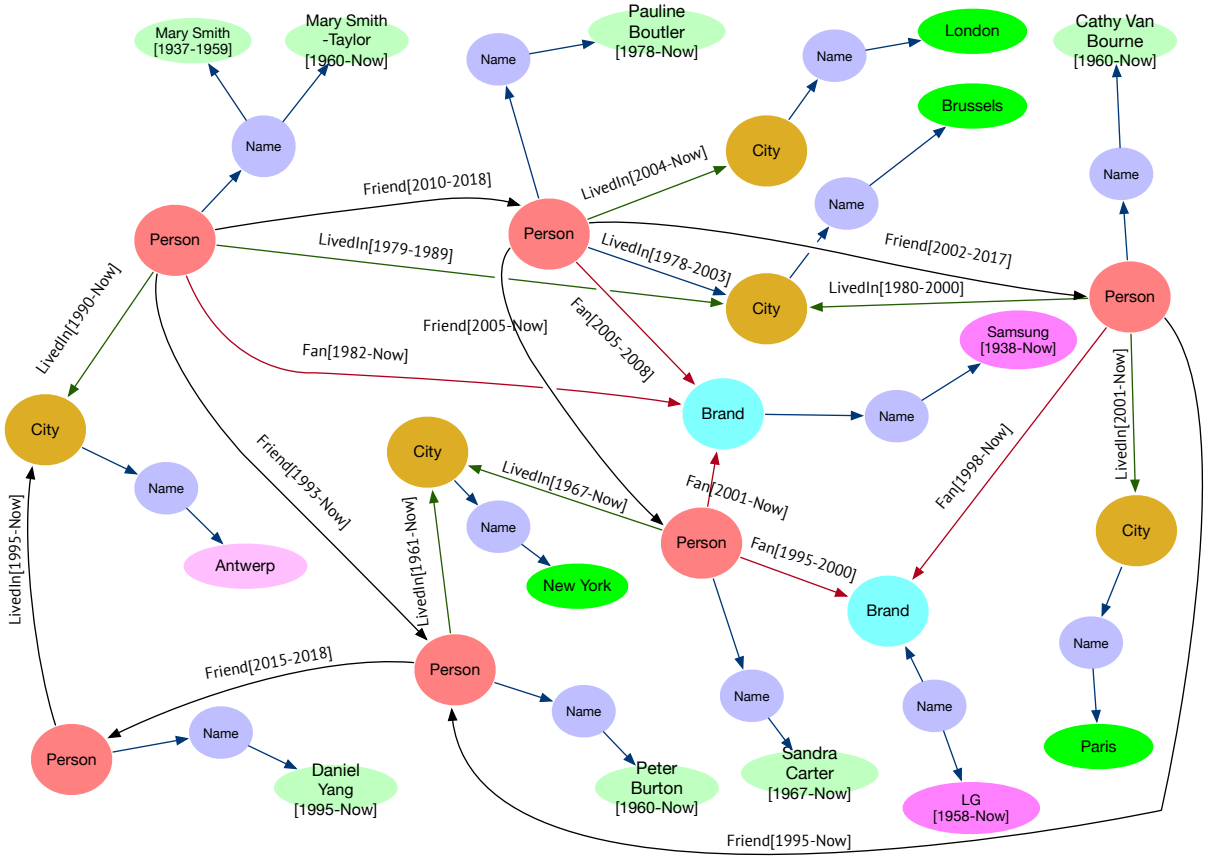

Fig. 2 A temporal graph and its different kinds of nodes.

12. $\forall n_a\{n\} \in N_a, n_a.interval \subset n.interval$
13. $\forall n_v\{n_a\} \in N_v, n_v.interval \subset n_v.interval$
14. $\forall n_v\{n_a\}, n'_v\{n_a\}, n_v \neq n'_v, n_v.interval \cap n'_v.interval = \emptyset$

Constraints 1 through 3 state that all nodes in the graph have a different id. Constraint 4 requires coalescing all nodes with the same value associated with the same attribute node; thus, the interval becomes a temporal element which includes all periods where the node had such value. Analogously, Constraint 5 applies to edges: all edges with the same name (i.e, representing the same relationship type), between the same pair of nodes, are coalesced. Constraints 6 through 8 state how the nodes must be connected, namely: (a) An Object node can only be connected to an attribute node or to another object node; (b) Attribute nodes can only be connected to non-attribute nodes; and (c) Value nodes can only be connected to attribute nodes. The cardinalities of these connections are stated by Constraints 9 through 10, which tell that attribute nodes must be connected by only one edge to an object node, and value nodes must only be connected to one attribute node with one edge. Finally, Constraints 11 to 14 restrict the values of the `interval` property. □

3.1 Continuous Path

In ILTGs, it is usually the case when queries ask for paths that are valid continuously during a certain interval. This requirement is captured by the notion of *continuous path* [43], introduced in Definition 7.

Definition 7 (Continuous Path) Given a temporal property graph G (interval-labeled), a *continuous path* (cp) with interval T from node n_1 to node n_k, traversing a relationship r, is a sequence $(n_1, \ldots, n_k, r, T)$ of k nodes and an interval T such that there is a sequence of consecutive edges of the form $e_1(n_1, n_2, r, T_1)$, $e_2(n_2, n_3, r, T_2)$, $\ldots$, $e_k(n_{k-1}, n_k, r, T_k)$, $T = \bigcap_{i=1,k} T_i$. □

Example 3 (Continuous Path) Consider the graph in Figure 3, where $e_1(n_1, n_2, friend, [1, 9])$, $e_2(n_2, n_3, friend, [2, 3])$, $e_3(n_3, n_4, friend, [1, 10])$, $e_4(n_1, n_5, friend, [2, 8])$, and $e_5(n_5, n_4, friend, [4, 7])$. There are two continuous paths, $(n_1, n_2, n_3, n_4, friend, [2, 3])$ and $(n_1, n_5, n_4, friend, [4, 7])$. That is, n_4 can be reached traversing the edges labeled *friend* from n_1 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

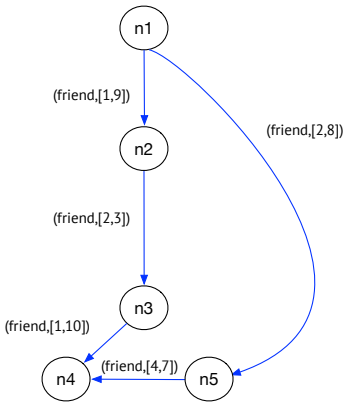

Fig. 3 Continuous paths.

n_4 is continuously reachable from n_1, is obtained by taking the union of both intervals, that is $[2, 7]$. □

3.2 Pairwise Continuous Path

Requiring a path to be valid throughout a time interval is a strong condition for a graph query. In many cases, querying temporal graphs requires a weaker notion of temporal path. Consider for example the case of a social network like the one in Figure 2. Also assume that there are friendship relationships between a person p_1 and a person p_2, in an interval $[2, 7]$. Also, p_2 was a friend of p_3 during the interval $[6, 12]$, and p_3 was a friend of p_4 during the interval $[10, Now]$. It can be seen that there is no continuous path from p_1 to p_4. However, the user may be interested in a transitive friendship relationship such that there is an intersection in the interval of two consecutive edges. In the example above such intersection exists, e.g., there is an overlap between $(p_1, p_2, friend, [2, 7])$ and $(p_2, p_3, friend, [6, 12])$, and between the latter and $(p_3, p_4, friend, [10, Now])$. That means, although there is not a continuous path between p_1 and p_4, there is a consecutive chain of pairwise temporal relationships. This is formalized by the notion of *pairwise continuous path*.

Definition 8 (Pairwise Continuous Path) Given a temporal property graph G, a *pairwise continuous path* between two nodes n_1, n_k, through a relationship r, is a sequence of edges $e1(n_1, n_2, r, [t_{s_1}, t_{f_1}]), \ldots, e_k(n_{k-1}, n_k, [t_{s_{k-1}}, t_{f_k}])$, such that $(ts_1 \leq ts_2 \leq tf_1 \vee t_{s_2} \leq t_{f_1} \leq t_{f_2}) \wedge \ldots \wedge (t_{s_{k-1}} \leq t_{s_k} \leq t_{f_{k-1}} \vee t_{s_k} \leq t_{f_{k-1}} \leq t_{f_k})$. □

3.3 Consecutive Paths

Figure 1 shows that DLTGs can also be represented as ILTGs. Therefore, the queries in Section 2.2.1, e.g., asking for earliest or fastest arrival times in a DLTG, require a different temporal semantics than the ones in Sections 3.1 and 3.2. Definition 9 introduces the notion of *consecutive path*.

Definition 9 A *consecutive path* P_c traversing a relationship r in a temporal property G is a sequence of edges $P = (e_1, e_2, r, [t_1, t_2]) \ldots, (e_{k-1}, e_k, r, [t_{k-1}, t_k]))$ where $(n_i, n_{i+1}, r, [t_i, t_{i+1}])$ is the i-th temporal edge in P for $1 \leq i \leq k$, and $t_{i-1} < t_i$ for $1 \leq i \leq k$. Instant t_k is the *ending time* of P, denoted $end(P)$, and t_1 is the *starting time* of P, denoted $start(P)$. The *duration* of P is defined as $dura(P) = end(P) - start(P)$, and the *distance* of P as $dist(P) = k$. □

With the notion of consecutive path, several different temporal paths can be defined, analogously to the paths for DLTGs described by Wu et al. in [52]. The ones studied in this paper are introduced in Definition 10.

Definition 10 (Types of consecutive paths) Let G be a temporal property graph G, a relationship r in G, a source node n_s, and a target node n_t, both in G; there is also a time interval $[t_s, t_e]$. Let $\mathcal{P}(n_s, n_t, r, [t_s, t_e]) = \{P \mid P$ is a consecutive path from x to y such that $start(P) \geq t_s$, $end(P) \leq t_e\}$. The following paths can be defined:
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. Formally,
EAP: $P \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])$ such that $end(P) = min\{end(P') : P' \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])\}$.
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. Formally,
LDP: $P \in \mathcal{P}(x, y, [t_s, t_e])$ such that $start(P) = max\{start(P') : P' \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])\}$.
The fastest (FP) is the path that can be completed in a given interval such that its duration is minimum. Formally,
FP: $P \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])$ such that $dura(P) = min\{dura(P') : P' \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])\}$.
The *shortest path (SP)* is the path that can be completed in a given interval such that its length is minimum. Formally,
SP: $P \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])$ such that $dist(P) = min\{dist(P') : P' \in \mathcal{P}(n_s, n_t, r, [t_s, t_e])\}$. □

Based on Definition 10, more kinds of paths can be defined to address practical problems. For example, for scheduling, a fastest path can be defined restricted to the paths such that there is a minimum 'waiting' time between two consecutive edges. Or, for phone fraud analysis, a path such that the time between two consecutive edges is below a given threshold, can be computed.

3.4 Updating the graph

A graph like the one in Figure 2 could be updated at any time. These updates may involve the addition or deletion of a node of any kind, and the addition or deletion of an edge. In addition, it is assumed that updates are performed over a consistent document (cf. Definition 6), and must leave the graph in a consistent state. Before discussing updates in more detail, the semantics of time must be further explained.

3.4.1 The semantics of time

In temporal databases, using a current time variable has several implications which require the definition of a precise semantics [15]. When using the transaction time approach, like in this work, the problems arising from the use of *Now* to represent a moving current time are considerably reduced compared with a valid time data model, because in valid time databases, timestamps are provided by the user, while in transaction time ones, these values are usually handled by the underlying database system. The semantics adopted for the current time variable *Now* in this work, is the one in [15], that is, if the ending point of an interval is *Now*, the edge is valid until the timestamped element is updated, yielding the so-called *until changed* semantics. As a consequence, the start instant of an interval can never be *Now*. In what follows, the attributes representing the start and end times of a time interval are denoted FROM and TO, respectively.

3.4.2 Node and edge updates

The *addition of a node* is straightforward. Adding an object node has no constraint. To add an attribute node, the corresponding object node must previously exist. Analogously, in the case of a value node, an attribute node must preexist. In all cases, the temporal constraints in Definition 6 require that, for example, the lifespan of an attribute node does not fall outside the lifespan of its associated object node, and the same for a value node with respect to an attribute node.

A deletion of a node or edge is performed in the temporal database sense. That means, only currently existing objects can be deleted. Informally, when deleting a node n at time t_d, *Now* is replaced by t_d in interval.TO. Analogously, updating an attribute or value node at time t_u implies deleting the current node at t_u, and creating a new one, where interval.FROM$= t_{u+1}$, where t_{u+1} is the instant immediately following the updating time in the node's granularity. The left-hand side of Figure 4 depicts an example. It shows that the name of Mary Smith was changed to Mary Smith-Taylor at time t_c. Now, assume that the complete Person node number 14 must be deleted at time t_d. This implies setting interval.TO$= t_d$ in nodes 14, 24, and 35. Thus, deleting an object node at time t_d also implies deleting in the temporal database sense, the related attribute and value nodes, that means, setting all *Now* values to t_d. Since consistency must be maintained, all currently incoming and outgoing edges must be 'deleted' too.

Adding or deleting an edge is a little bit more involved, since it impacts on the paths defined in Sections 3.1 through 3.3. The example on the right-hand side of Figure 4 shows that, for instance, the edge in dashed line can be inserted at any time, provided that the temporal constraints are satisfied. That means, in this example, that the edge interval could be anyone starting in 1998, the start time of the interval of Node 90. This shows that the model supports also a restricted form of valid time, since these kinds of retroactive updates are also allowed. Note, however, that this new edge produces a new continuous path with interval [2006, Now]. Now, assume that the person with id=14 stops following the person with id=55 at time t_d. Since edge deletion is also logical rather than physical, the interval of the edge between the two nodes becomes $[2005, t_d]$, and all continuous paths must be modified, since the edge ceases to exist. However, the continuous paths existing prior to t_d must remain. This impacts the indices that may exist over the paths (see Section 7). Also, note that if this person, after some time, starts following again the same person with id=55, at time t_i, a new interval must be added to the same edge, which becomes $\{[2005, td], [t_i, Now]\}$.

4 T-GQL Syntax and Semantics

This section introduces T-GQL, a high-level query language for graph databases. The language has a slight SQL flavor, although it is based on Cypher[4], Neo4j's high-level query language. Cypher's formal semantics can be found in [20,21]. T-GQL also extends Cypher with a collection of functions, whose implementation is explained in Section 5.

4.1 Basic Statements

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

[4] https://neo4j.com/docs/cypher-manual/current/

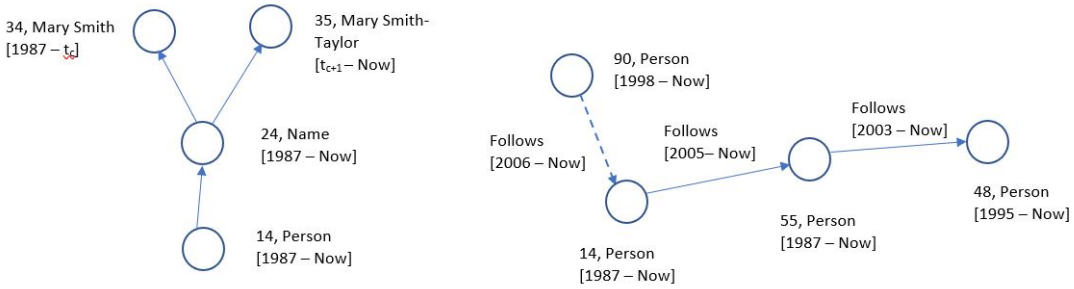

Fig. 4 Left: Updating nodes; Right: Updating edges.

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 will be introduced using the social network in Figure 2. Path functions implementing the consecutive path semantics will be covered using a flight scheduling example.

Consider the query: *"List the friends of the friends of Mary Smith-Taylor"*. This does not include temporal features, but allows introducing the basic T-GQL syntax.

```
SELECT p2
MATCH (p1:Person) - [:Friend*2] -> (p2:Person)
WHERE p1.Name = 'Mary Smith-Taylor'
```

Note that this query just returns the object nodes (recall the model of Definition 4), which, for a final user, would not be useful. A variant to the query above would select the name of the friends of friends of Mary as follows (an alias is used in the query):

```
SELECT p2.Name as friend_name
MATCH (p1:Person) - [:Friend*2] -> (p2:Person)
WHERE p1.Name = 'Mary Smith-Taylor'
```

For returning all the paths, the wildcard operator '*' is used. The expression below returns the three paths of length 2 from the node representing Mary.

```
SELECT *
MATCH (p1:Person) - [:Friend*2] -> (:Person)
WHERE p1.Name = 'Mary Smith-Taylor'
```

The T-GQL language supports the three path semantics explained in previous sections: (a) *Continuous path semantics*; (b) Pairwise continuous path semantics; (c) Consecutive path semantics. These semantics are implemented by means of functions, which are included in a library of Neo4j plugins. To compute temporal paths, two types of functions are defined: Coexisting and Consecutive. Both receive two nodes as arguments. These are explained in the following sections.

Remark 1 Functions computing continuous and pairwise paths, do not accept the wildcard '*'. That is, the length of the paths must be constrained by the user. On the contrary, temporal functions computing consecutive paths (earliest, fastest, etc.), do not support a limited search, therefore '*' must be used.

4.2 Continuous Path Queries

Query 1 requires the computation of all continuous paths of length 2, over the social network running example. As Remark 1 mentions, the length of the continuous paths in a query must be explicitly specified.

Query 1 *Compute the friends of the friends of each person, and the period such that the relationship occurred through all the path.*

In Figure 2, for example, Cathy (person node 12) was a friend of Pauline (person node 11) between 2002 and 2017. Also, Pauline was a friend of Mary (person node 14) between 2010 and 2018. Thus, the path ($Mary \rightarrow Pauline \rightarrow Cathy, [2010, 2017]$) will be in the answer to Query 1, since the whole path was valid in this interval (Definition 7). The query reads in T-GQL:

```
SELECT path
MATCH (n:Person), path = cPath((n)-[:Friend*2]
       -> (:Person))
```

In this case, a record is returned for each path. The modifiers SKIP and LIMIT can be used, as in Cypher, to get a specific path or a range. For example, to get the third path in the answer:

```
SELECT path
MATCH (n:Person), path = cPath((n)-[:Friend*2]
       -> (:Person))
SKIP 2
LIMIT 1
```

A continuous path search between two specific persons can also be performed, as Query 2 shows.

Query 2 *Find the continuous paths between Mary Smith Taylor and Peter Burton 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 = 'Mary Smith-Taylor'
      and p2.Name = 'Peter Burton'
```

The cpath function computes the continuous path. The result is a single path of length three (the other possible path, with length one, is discarded). The path is an array of the object nodes traversed together with their interval, attributes, id and title. The interval of the result is the intersection of the intervals of the object nodes in the path.

The figure below shows the format of the result. It can be seen that attribute and value nodes are embedded in the answer in an inline fashion, to facilitate their search (as mentioned previously, object nodes are not likely to be useful for a final user). Note that the value node "Mary Smith" is ignored since its interval [1937-1959] does not intersect with the continuous path's interval $[2010 - 2017]$. Also note that the value node returned has the interval $[2010 - 2017]$, which is the intersection of the intervals $[1960 - Now]$ (the interval of the value) and $[2010 - 2017]$. Finally, the interval of the continuous path is $[2010 - 2017]$, which is the result of the intersection between the traversed edges $([2010 - 2018], [2002 - 2017], [1995 - Now])$.

paths

```
{
"path": [
    { "interval": ["1937-Now" ],
      "attributes": {
        "Name": [
          {"value": "Mary Smith-Taylor",
            "interval": "[2010 - 2017]" }]
         },
      "id": 8,
      "title": "Person"
    },
    {
      ...
    }
  ],
  "interval": "2010-2017"
}
```

The cPath function is overloaded to return a Boolean value, like Query3 shows.

Query 3 *Find the names of the persons such that there is a continuous path from them to Peter Burton.*

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

In this case the function call is located in the WHERE clause, and the parser decides from the context that the Boolean procedure must be used.

Pairwise continuous paths (Definition 3.2) can be also computed, using the pairCPath function. An example is shown below.

Query 4 *Find the pairwise continuous paths between Mary Smith Taylor and Peter Burton with a minimum length of two and a maximum length of three.*

```
SELECT paths
MATCH (p1:Person), (p2:Person),
paths = pairCPath((p1)-[:Friend*2..3]->(p2))
WHERE p1.Name = 'Mary Smith-Taylor'
      and p2.Name = 'Peter Burton'
```

The intermediate results of a query can be filtered by an interval I, provided by the user. This filters out the paths whose interval does not intersect with I. The granularity of the starting and ending instants of the interval must be the same. Query 5 illustrates this.

Query 5 *Compute all the continuous paths of friends between Mary Smith Taylor and Peter Burton, in the interval* $[2018, 2020]$*, with a minimum length of 2 and maximum length of three.*

In the running example, there are two possible paths between Mary and Peter: one of length 3 and the other of length 1 (which is thus, discarded). Therefore, the only continuous path obtained would be $Mary \rightarrow Pauline \rightarrow Cathy \rightarrow Peter, [2010, 2017]$. However, the path will be filtered out of the result set, since $[2018, 2020] \cap [2018, 2020] = \emptyset$. The query is expressed as:

```
SELECT paths
MATCH (p1:Person), (p2:Person),
paths = cPath((p1) - [:Friend*2..3] -> (p2),
        '2018', '2020')
WHERE p1.Name = 'Mary Smith-Taylor'
      and p2.Name = 'Peter Burton'
```

The properties of the returned structure can also be retrieved. For example, if only the interval of the path is needed in Query 2, the query would read:

```
SELECT paths.interval as interval
MATCH (p1:Person), (p2:Person),
```

```
paths = cPath((p1) - [:Friend*2..3] -> (p2))
WHERE p1.Name = 'Mary Smith-Taylor' and
      p2.Name = 'Peter Burton'
```

Furthermore the attributes in the path can be retrieved as in the following query, where the names of the persons in the starting and in the the third position in the resulting paths are requested.

```
SELECT paths.path[0].attributes.Name
       as start_node,
       paths.path[3].attributes.Name as
       end_node
MATCH (p1:Person), (p2:Person),
paths = cPath((p1) - [:Friend*2..3] -> (p2))
WHERE p1.Name = 'Mary Smith-Taylor' and
      p2.Name = 'Peter Burton'
```

The head() and last() path methods can be used as follows.

```
SELECT head(paths.path).attributes.Name as
  start_node, last(paths.path).attributes.Name
  as end_node
MATCH (p1:Person), (p2:Person),
paths = cPath((p1)-[:Friend*2..3]->(p2))
WHERE p1.Name = 'Mary Smith-Taylor' and
      p2.Name = 'Peter Burton'
```

If more than one path were returned, the head() and last() functions will be applied to each one.

4.3 Consecutive path queries

To illustrate consecutive path semantics (Definitions 9 and 10), a second running example is introduced, depicted in Figure 5. In this example, there are two object nodes, namely Airport and City. There are also two temporal relationships, Flight and LocatedAt. The former is labeled with the interval $[t_d, t_a]$, where t_d is the departure time of a flight from an airport, and t_a is the arrival time at the destination airport. Airport nodes are labeled with the period during which an airport belongs to a city (not shown in the figure, for clarity). Note here the flexibility that the ILTG model provides, allowing representing cases that are typically modeled using DLTGs. It is worth remarking that, of course, this does not intend to be a real-world example of a flight scheduling graph, but a simplified portion of it.

Consecutive path semantics is implemented through functions that are called from T-GQL. Four functions are currently supported: fastestPath, earliestPath, shortestPath, and latestDeparturePath. The first three ones receive two nodes as arguments. The latter also receives a time instant. The queries below illustrate their syntax and semantics.

Query 6 *How can we go from Tokyo to Buenos Aires as soon as possible?*

Recalling Definition 10, Query 6 refers to the earliest-arrival path from Tokyo to Buenos Aires. Note that this query uses the consecutive path semantics of Definition 9. Here, the difference with the continuous path semantics is clear: a path in the solution must be such that the intervals of the edges are pairwise disjoint. 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 = 'Tokyo'
```

Opposite to the earliest-arrival path function, the latestDeparturePath function needs a threshold parameter as argument. As an example, consider Query 7 below.

Query 7 *How can we go from Tokyo to Buenos Aires, leaving as late as possible and arriving before July 15 at 8 pm?*

```
SELECT path
MATCH (c1:City)-[:LocatedAt]->(a1:Airport),
      (c2:City)-[:LocatedAt]->(a2:Airport),
path = latestDeparturePath((a1)-[:Flight*]->
       (a2),'2019-07-15 20:20')
WHERE c1.Name='Buenos Aires' AND
      c2.Name='Tokyo'
```

4.4 Handling Temporal Granularity

The reader may have noticed that all time intervals in the social network example are given in the Year time granularity; for the flight example, granularity is Datetime. However, queries may mention a granularity different to the one in the graph's objects. This time granularity problem has been extensively studied in temporal database theory, and it is common to all kinds of queries. When a query includes a temporal condition with a temporal granularity t_g different than the one of an object in the graph o_g, two cases may occur:

- t_g is finer than o_g. In this case, both granularities are identified, in a way such that the finer one is transformed into the coarser one. For example, if $o_g.interval = [2010, 2012]$, and the condition is $t\ IN\ o_g.interval$, where $t = 2/10/2012$, then, the interval is transformed into the interval $o_g.interval = [1/1/2010, 31/12/2012]$.

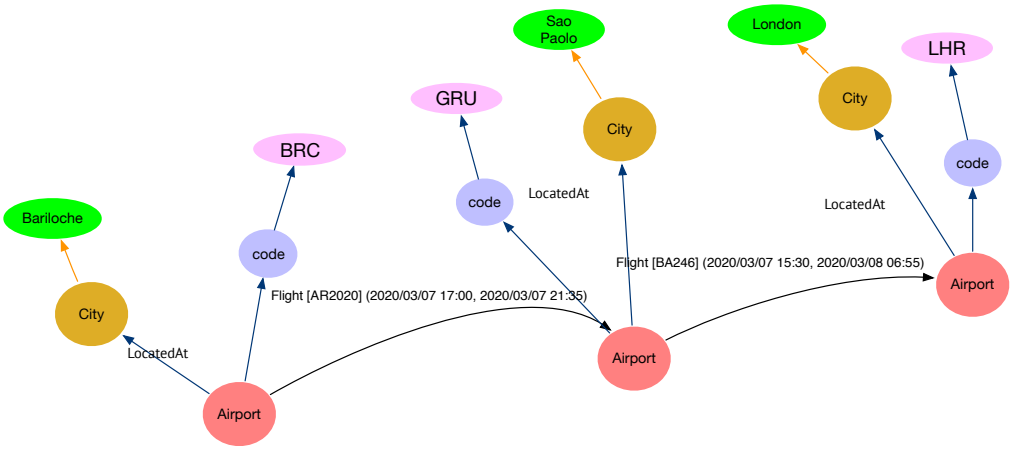

Fig. 5 A temporal graph for flight scheduling analysis.

– t_g is coarser than o_g. In this case, one time instant in the granularity of o_g is chosen. For example, if $o_g.interval = [15/10/2010, 23/12/2010]$, and the condition is 2010 IN $o_g.interval$, the semantics would imply that the condition is satisfied.

Since in the social network example, the granularity used is Year, and the example queries are given using this granularity, no problem arises in this sense. However, if a query asks for Cathy's friends on October 10th, 2018, it would not be possible to give a precise answer, and the query must use the semantics explained above. T-GQL supports the following granularities (examples will be presented in the next sections):

– Year: yyyy
– YearMonth: yyyy-MM
– Date: yyyy-MM-dd
– Datetime: yyyy-MM-dd HH:mm

4.5 Temporal Operators

Some kinds of T-GQL queries require temporal operators and filters, explained in this section. To begin with, the SNAPSHOT operator returns the state of the graph at a certain point in time. Therefore, along the lines of temporal database notions, the answer is a non-temporal graph, like in Query 8 below.

Query 8 *Who where the friends of the friends of Cathy in 2018?*

```
SELECT p2.Name as friend_name
MATCH (p1:Person) - [:Friend*2] -> (p2:Person)
WHERE p1.Name = 'Cathy Van Bourne'
SNAPSHOT '2018'
```

Exactly one value is allowed to be used in the SNAPSHOT clause. The following non-temporal result is returned:

p2.Name
{ "value": "Mary Smith Taylor" }

The relationship with Pauline is filtered out since it was valid during the interval $[2002, 2017]$. Therefore, there is only one object node reached, which has two possibles values for the Name attribute. The value "Mary Smith" is discarded because it was not valid in 2018.

The BETWEEN operator performs an intersection of the graph intervals with a given interval. Exactly one interval is allowed. The granularity of both intervals must be the same, like in Query 9 below.

Query 9 *Where did the friends of Pauline live between 2000 and 2004?*

This query returns the cities where the friends of Pauline lived during the given interval. The temporal semantics adopted also applies the condition on the relationship interval. That means, for example, that the relationship with Sandra will not be considered, since the interval of the relationship is $[2005, Now]$, thus, it does not intersect with the given interval. The T-GQL query is written as follows:

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

Only the Friend relationship with Cathy Van Bourne was valid during the interval used above, and the query returns Brussels and Paris, the cities where she lived during the intervals $[1980, 2000]$, and $[2001, Now]$, the ones that intersect $[2000, 2004]$.

Finally, the WHEN clause is useful for answering parallel-period queries, which follows the SQL inner

query idea. The syntax has the form MATCH-WHERE-WHEN, and the inner query can have references to variables in the outer query. Function calls are not allowed within this clause, and it can only handle exactly one two-node path in the inner MATCH clause.

Query 10 *Who were friends of Mary while she was living in Antwerp?*

Mary lived in Antwerp between [1990-*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. Query 11 shows an even more involved example.

Query 11 *Where did Cathy live when she and Sandra followed the same brands?*

Cathy and Sandra both followed the brand *LG*. Sandra, during the interval [1995, 2000], and Cathy, in the interval [1998-2000]. The query language allows expressing a graph traversal to the node that indicates where did Cathy live from 1998 to 2000. In this case, it would be the city of Brussels. For this, the query must compute the intersection of the intervals. Note that the former two queries would be much difficult and unnatural to express with a duration-labeled representation.

```
SELECT c.Name as city, b1.Name as brand
MATCH (p1:Person) - [:LivedIn] -> (c:City),
(p1) - [:Fan] -> (b1:Brand)
WHERE p1.Name = 'Cathy Van Bourne'
WHEN
  MATCH (p2:Person) - [f:Fan] -> (b2:Brand)
  WHERE p2.Name = 'Sandra Carter' and
        b1.Name = b2.Name
```

5 Implementation

This section describes a proof-of-concept implementation of this proposal. First, the general system architecture is presented. Then, the parsing process and the translation of a T-GQL query to Cypher are explained. Finally, the algorithms for computing the temporal operators and the different kinds of paths are discussed.

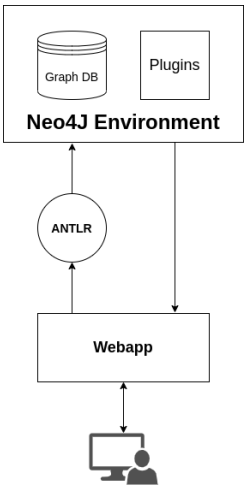

Fig. 6 General architecture.

5.1 Architecture

The model and language described in this paper were implemented over the open source Java-based graph database Neo4j. Neo4j allows extending its functionality with user-defined procedures, which can be easily added as plugins, packed in a .jar file. These procedures can then be used in Cypher queries as any of the other built-in functions that this language offers.

The T-GQL language grammar was implemented using ANTLR.[5] With this tool, T-GQL queries are translated into Cypher, Neo4j's high-level query language, so it can be executed over the Neo4j database. Figure 6 sketches the system's architecture. To edit and execute T-GQL queries, a web application interface was developed, also coded in Java, using the Javalin framework.[6] The application exposes a page where the queries can be executed from an endpoint. The parser translates the users' queries into Cypher and executes them on a Neo4j server that contains the plugins to run the temporal operators and path algorithms.

In addition, for populating the social network running example database (and also for the experiments reported in the next section), a data set generator was developed. Parameters for this generator allow indicating the number of relationships and nodes, and number of intervals that each edge can have, among other ones. The application communicates directly with a running Neo4j server through the Bolt protocol, and automatically populates the database by executing the corresponding Cypher queries.

[5] https://www.antlr.org/
[6] http://javalin.io

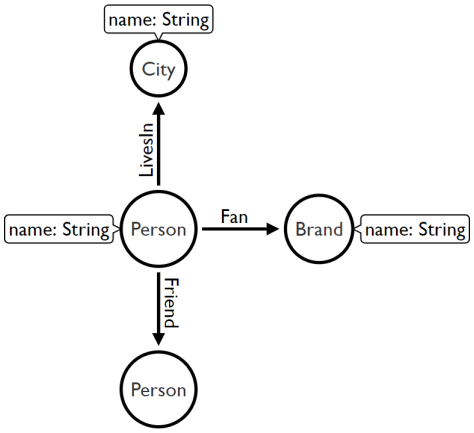

Fig. 7 Social network metamodel.

5.2 Parsing and Query Translation

The parser was developed using ANTLR4, a parser generator that reads a grammar and produces a recognizer for it. It is important to keep in mind that the query language hides the actual data structure of the graph. Recall that the model explained in Section 3 is composed of three kinds of nodes, namely object, attribute, and value nodes, but the user writes her queries abstracting from these elements. Consider for instance, the metamodel of the social network example, depicted in Figure 7. It can be seen that Person, City, and Brand are object nodes, connected by different kinds of relationships. These object nodes are associated with attribute and value nodes through a single kind of edge, denoted Edge (also not visible to the user). Thus, in the implementation, Person is actually a property (denoted title) of the object node, the Name of a person is a property (also denoted title) of an attribute node, and the actual name of the person is stored as a property of a value node, denoted value. All of these elements, again, are not perceived by the user, but stored in the Neo4j database, as shown in Figure 8. In the figure it can be seen that there is an edge labelled Edge outgoing from an object node labeled Person (which is the value of the property title of the object node). That edge reaches the attribute node Name (again, Name is a property of the attribute node), and finally another Edge links that node with a value node with value = 'New York'. Note that all of these nodes and edges are associated with intervals, not shown in the figure. The translation, then, must not only rewrite the query in terms of the Cypher language, and bridge the gap between the structure exposed to the user, and the model actually stored in Neo4j.

To illustrate the parsing process, consider the query:

```
SELECT p
MATCH (p:Person)
WHERE p.Name = 'John Smith'
```

Figure 9 depicts the parse tree. The start rule is highlighted in blue, non-terminal nodes are indicated in yellow, and terminal nodes in green. For the sake of simplicity, not all the nodes needed for evaluating this query are expanded and represented in the tree. Once the tree has been generated, it must be traversed. ANTLR's default method is represented in the figure in dashed line. First, all the tokens in the SELECT clause are recognized, followed by the MATCH clause, and finally the WHERE clause. When the tree is fully traversed, the Cypher query is generated. The query translation process is explained next.

The object nodes in the MATCH clause are translated as {alias:Object {title: 'Name'}}, since, as explained above, this property contains the entity type that the user refers to. For example "(p:Person)" would be translated to {p:Object {title: 'Person'}}. The edges do not need to be translated, since the grammar for the edges matches the Cypher's grammar. If a function call is found, the corresponding procedure is called, with the given arguments (an example of this is shown later). For each attribute in the SELECT clause, a three-node path (Object - Attribute - Value) is produced from the object node. For example "p.Name as name" would generate the following path:

```
OPTIONAL MATCH(p)-->(internal_n:Attribute
            {title:'Name'})-->(name:Value)
```

Recall that title is a property of the attribute node. In this case, OPTIONAL MATCH is used to allow replacing the missing values in the SELECT clause with a NULL value, and to return the row instead of discarding it. Variables starting with 'internal' are generated internally by the parser, and are reserved. For the conditions in the WHERE clause, the attributes are expanded as explained above, and the constants are translated without changing them. Finally, for each attribute, the access to the value property of the value node, is added. For example, the condition "p.Name = 'John' and p.Age = 18" is translated as:

```
MATCH (p)-->(internal_n:Attribute{title:
      'Name'})-->(internal_v:Value)
MATCH (p)-->(internal_a:Attribute{title:
      'Age'})-->(internal_v1:Value)
WHERE internal_v.value = 'John' and
      internal_v1.value = 18
```

Queries mentioning functions are explained next. Consider the continuous path query:

```
SELECT p.path as path, p.interval as interval
```

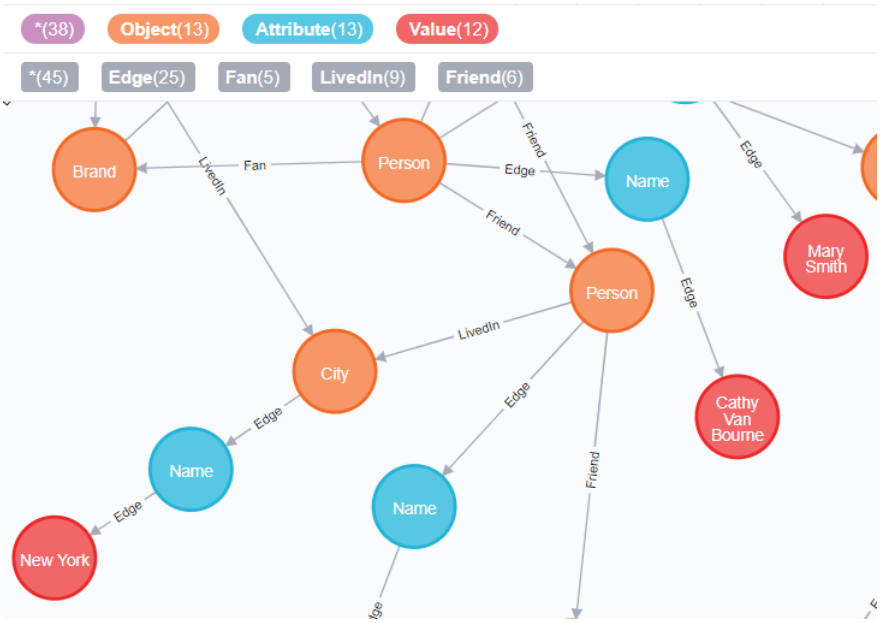

Fig. 8 Social network model for the metamodel in Figure 7.

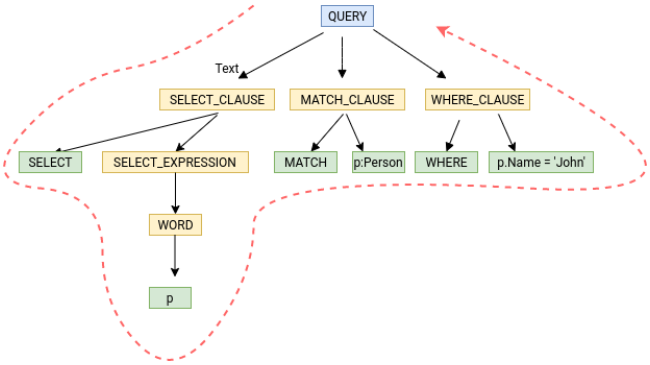

Fig. 9 Example parse tree.

```
MATCH (p1:Person), (p2:Person), p=cPath((p1)-
      [:Friend*2..3]->(p2),'2016','2018')
WHERE p1.Name = 'Mary Smith-Taylor'
```

The query is translated into Cypher as:

```
MATCH (p1:Object{title:'Person'}),(p2:Object
      {title:'Person'})
MATCH (p1)-->(internal_n0:Attribute{title:
             'Name'})-->(internal_v0:Value)
WHERE internal_v0.value = 'Mary Smith-Taylor'
CALL coexisting.coTemporalPaths(p1,p2,2,3
  {edgesLabel:'Friend',nodesLabel:'Person',
  between:'2016-2018',direction:'outgoing'})
  YIELD path as internal_p1, interval as
  internal_i1
WITH {path:internal_p1,interval:internal_i1}
      as p
RETURN p.path as 'path', p.interval as
       'interval'
```

Temporal procedures are described in Section 5.3. Note that after calling these path procedures, the query may ask for just one of the computed paths. For example, the following query asks for the fastest path between airports located in the cities of London, UK and Bariloche, Argentina, both with more than one airport.

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

This is translated to:

```
MATCH (c1:Object{title:'City'})<-[internal_l0:
      LocatedAt]-(a1:Object{title:'Airport'}),
      (c2:Object{title:'City'})<-[internal_l1:
      LocatedAt]-(a2:Object{title:'Airport'})
MATCH (c1)-->(internal_n0:Attribute{title:
      'Name'})-->(internal_v0:Value)
MATCH (c2)-->(internal_n1:Attribute{title:
      'Name'})-->(internal_v1:Value)
WHERE internal_v0.value='London' AND
      internal_v1.value='Bariloche'
CALL consecutive.fastest(a1,a2,1,
    {edgesLabel:'Flight',direction:'outgoing'})
YIELD path as internal_p0, interval as
      internal_i0
WITH paths.intervals.fastest({path:internal_p0,
     interval:internal_i0}) as path
RETURN path
```

To evaluate this Cypher query, the engine will look for all the airports in London and Bariloche, and all the combinations from airports in London to airports in Bariloche. The `paths.intervals.fastest` aggregation function is called to retrieve the fastest path. It receives all the paths and returns only the fastest ones, according to Definition 10.

5.3 Temporal Procedures Algorithms

It was already explained that the Neo4j database was extended with temporal capabilities by means of a collection of procedures. Implementing the procedures on the server side allows calling the procedures directly from the Cypher Language. Besides, a client-side implementation would require retrieving a large portion of the graph to execute the queries, which would not scale for large graphs. Thus, the algorithms will use less resources running on the server side, since nodes and relationships are obtained directly from the database. Procedures can be classified in three groups, depending on their functionality:

- *Temporal procedures:* Implement basic temporal operations. Here, Between and Snapshot are defined.
- *Coexisting paths procedures:* Implement the continuous and pairwise continuous path semantics.
- *Consecutive paths procedures:* Implement the consecutive path semantics.

The procedures above are packed in a library which is stored in the Neo4j's Plugin folder. The *Coexisting* and *Consecutive* procedures extend a framework defined to work on temporal graphs. This framework was based on the neo4j-graph-algorithms library,[7] which contains implementations of classic graph algorithms, although no algorithms for temporal graphs.

5.3.1 Temporal procedures

The Between and Snapshot procedures receive a Cypher query, execute it, and filter the results depending on the operation. Neo4j returns the results of a query as a stream of records, analogously to relational databases. The operations above are thus applied to all the rows in the stream, filtering the results that do not satisfy the temporal restrictions. In both cases, the procedure receives a string containing the query, and another string representing the granularity that must be applied to the operation. In addition, the Between operation receives an interval, and keeps the records in the stream whose intervals are inside the former one. The Snapshot operation also receives a string that contains a specific time instant, and keeps the records whose intervals contain that specific time instant.

5.3.2 Coexisting paths procedures

These procedures return the continuous paths of a given length, either starting from a node, or between two nodes. In addition, a Boolean alternative is implemented, that can be used, for example, for checking whether or not a continuous path exists between two nodes.

Algorithm 1 retrieves all of the coexisting paths between two nodes, receiving as input a graph G, a source node x, the minimum path length L_{min}, the maximum path length, L_{max}, a function f that returns an interval depending on the algorithm, and optionally, a destination node y. The algorithm returns a set S with the results. Given two intervals, the function f returns another interval. When computing continuous paths, f is defined as $f(i1, i2) = i1 \cap i2$. This way, only the intersection of the intervals is stored, and the algorithm keeps iterating with them. For pairwise temporal paths, f is defined as $f(i1, i2) = i2$, this way it only returns the latter interval, and the algorithm iterates only with the last interval in the path.

The algorithm takes the source node x and adds it to a list, in a triplet containing an interval $[-inf, +inf]$, whose values are the minimum and maximum time instants of the node, and the length of the path, initially set to zero. This list represents a path that starts at the source node, and is added to the queue. The algorithm picks up the paths in the queue until the queue is empty. The algorithm takes the last triplet of the path, and looks up in the graph G for the edges associated with the node in this triplet. Then, for each edge, it checks if the node in the opposite end of the edge is in the path, or the interval in the edge does not intersect with the interval in the triplet. If that is the case, the edge cannot continue the path. This prevents iterating over the same nodes. For example, given an edge from A to B with interval [1, 2], a path A-B-A-B would be possible without this limitation, because the interval between A and B always intersects with itself. In the case that the edge can continue the path, a triple with the new node is created, containing the result of the execution of the function f, and the length of the path, which is the length of the last triplet in the path, plus 1. The path is copied and the triplet is added to the copy. If the copy of the path (which is also a path) has a length between L_{min} and L_{max} and the node of the last triplet is also the destination node (if such node is defined as input), this path is added to the set of solu-

[7] `https://github.com/neo4j-contrib/neo4j-graph-algorithms`

Algorithm 1 Computes Coexisting Paths (Continuous and pairwise continuous paths).

Input: A graph G, a source node x, the minimum path length L_{min}, the maximum path length L_{max}, a function f depending on the type of path requested, and a destination node y (optional).
Output: A list of coexisting paths S.

```
Initialize a queue of paths Q and a list of solutions S.
Q.enqueue([(x, [−inf, +inf], 0)])
while not Q.isEmpty do
    current = Q.dequeue()
    z, interval, length = current.last()
    for (z, otherInterval, dest) ∈ G.edgesFrom(z) do
        if not current.containsNode(dest) and interval ∩
        otherInterval ≠ ∅ then
            newTuple = (dest, f(interval, otherInterval), length +
            1)
            copy = current.copy()
            copy.insert(newtuple)
            if Lmin <= length + 1 <= Lmax and (y not exists or
            dest == y) then
                S.insert(copy)
            end if
            if length + 1 < Lmax then
                Q.enqueue(copy)
            end if
        end if
    end for
end while
```

Algorithm 2 Checks the existence of a Continuous Path.

Input: A graph G, a source node x, the minimum path length L_{min}, the maximum path length L_{max}, a function f which depends on the type of path requested (continuous o pairwise), and a destination node y (optional).
Output: *True* if a Continuous Path exists. *False* otherwise.

```
Initialize a queue of paths Q.
Q.enqueue([(x, [−inf, +inf], 0)])
while not Q.isEmpty do
    current = Q.dequeue()
    z, interval, length = current.last()
    for (z, otherInterval, dest) ∈ G.edgesFrom(z) do
        if not current.containsNode(dest) and interval ∩
        otherInterval ≠ ∅ then
            newTuple = (dest, f(interval, otherInterval), length +
            1)
            if Lmin <= length + 1 <= Lmax and (y not exists or
            dest == y) then
                return true
            end if
            if length + 1 < Lmax then
                copy = current.copy()
                copy.insert(newtuple)
                Q.enqueue(copy)
            end if
        end if
    end for
end while
return false
```

tions S. Otherwise, it is added to the queue. When this queue is empty, the set of solutions S is returned.

Algorithm 2 is the Boolean version of the previous one, since it computes if there exists a continuous path between two nodes. That is, if a path is found, *true* is returned, otherwise, it returns *false*.

5.3.3 Consecutive paths procedures

These procedures follow the graph transformation approach introduced by Wu et al. [54] for DLTGs, to compute paths on ILTGs. However, unlike the approach presented in [54], the algorithm presented here does not create the whole graph to apply the path computation algorithms, since this would be extremely expensive. Instead, the transformed graph is built as the iterations proceed over the original temporal graph, call it G. The transformation creates a new graph, denoted G_t, where the nodes contain either the starting time or the ending time of an interval of the temporal graph (explained below), and the edges indicate the nodes that are reachable from that position, where reachable means that both nodes are included in the same interval, or that they start from the same node and the starting time of the source node is prior to the one in the destination node. The weight of an edge is the duration of the corresponding interval. This new graph does not contain cycles, because it is not possible to go from a node with a greater time to a node with a lesser time, and all the weights of the edges are (or can be represented as) positive numbers.

Algorithm 3 sketches the process. The algorithm receives, as arguments, a temporal graph G, the source and destination nodes of the path (s and d, respectively) to be computed, a function f to be used to sort the nodes of the transformed graph in a priority queue -in a way which depends on the algorithm (earliest, latest, fastest, shortest paths), and returns a set of nodes S. The following is assumed in the sequel for f:

$$x < y \ if \ f(x,y) < 0$$
$$x = y \ if \ f(x,y) = 0$$
$$x > y \ if \ f(x,y) > 0$$

The nodes of the transformed graph G_t have four attributes: a reference to the node in the original graph, a time instant, the length of a path that passes through that node to iterate the graph in a DFS way, and a reference to the previous node in G_t, in order to allow rebuilding the paths after running the algorithm. These attributes are denoted (for a node n), $n.noderef$, $n.time$, $n.length$ and $n.previous$ in Algorithm 3.

After initializing the necessary structures, the algorithm adds the initial transformed graph node to the priority queue. This node is a quadruple that contains the source node s, $-\infty$ as the time instant, 0 as length, and *null* as the reference to the previous node. An element e is picked up from the queue until the queue is empty. There is a node v_i in the temporal graph associated with e. For each edge outgoing from v_i in G, the node is expanded creating the nodes v_{out} and v_{in} in the transformed graph. The node v_{out} contains the current node v_i, the start time of the interval in the edge, the length of e plus 1, and e as the previous node, that means $(v_i, t.start, e.length+1, null)$. The node v_{in} con-

Algorithm 3 Compute the minimum consecutive paths.

Input: A graph G, a source node s, a destination node d. A comparison function $f(x, y)$ where $x < y$ if $f(x, y) < 0$.
Output: A set with the optimal solutions S.

```
Initialize the transformed graph G_t and Q (priority queue of G_t nodes)
Q.enqueue((s, −∞, 0, null))
while not Q.isEmpty do
    current = Q.dequeue()
    for (current.node, interval, dest) ∈ G.edgesFrom(current.node) do
        if current.time > interval.start then
            continue
        end if
        vOut = (current.node, interval.start, current.length + 1, current)
        vIn = (dest, interval.end, current.length + 1, vOut)
        if G_t.containsNode(vIn.node, vIn.time) then
            othervIn = G_t.get(vIn.node, VIn.time)
            if f(othervIn, vIn) > 0 then
                continue
            end if
        end if
        if dest == d then
            if S.isEmpty then
                S.add(vIn)
            else
                s = S.getAny()
                comp = f(vIn, s)
                if comp > 0 then
                    S.empty()
                    S.add(vIn)
                else if comp == 0 then
                    S.add(vIn)
                end if
            end if
            continue
        end if
        Q.insert(vIn)
    end for
end while
return S
```

tains the destination node of the edge, the end time of the interval in the edge, the length of e plus 1, and v_{out} as the previous node, that is $(v_f, t_{end}, e.length{+}1, v_{out})$. If the start time of the interval is less than the time instant of e, the path is not expanded, because it means that this interval occurred prior to the interval associated with the instant. For example, for the interval $[5, 8]$, if the time instant in e is 7, the node will not be expanded, and it would not yield a consecutive path.

After creating v_{in} and v_{out} in the transformed graph G_t, the algorithm checks if G_t already contains a node $v_{in'}$ such that the temporal graph node and the time moment are the same as the ones in v_{in}. If this is the case, the two nodes are compared with the function f. If $f(v_{in}, v_{in'}) < 0$, the path is discarded. If $f(v_{in}, v_{in'}) > 0$ the node is replaced. Otherwise, the node is kept in the graph. The rationale behind discarding the paths is that if two paths P_1 and P_2 in G_t that end at the same node d, contain the same transformation node n, if $f(P_1(n), P_2(n)) > 0$, then $f(P_1(d), P_2(d)) > 0$, since the same nodes will be expanded, and the function f depends on the nodes already traversed (e.g., for the shortest-path, f depends on the path length, for the earliest-path, it depends on the arrival time to each node, and so on). Then, if v_i, the temporal graph node in v_{in} is not the same as the one in the destination node d, v_{in} is added to the queue. If v_i is the same as in d, and $S = \emptyset$, v_{in} is added to S. If $S! = \emptyset$, then any $s \in S$ is picked up. If $f(s, v_{in}) < 0$, the whole set S is discarded $f(v_{in}, s) == 0$, v_{in} is added to S, and if $f(v_{in}, result) > 0$, S is reset to $\{v_{in}\}$. When Q is emptied, the set of nodes in G_t is returned, and the algorithm reconstructs the paths using the stored references to previous nodes in the paths. That is, for each node, the algorithm follows the link to the previous node until there is no previous node, like in the implementation of the Dijkstra algorithms.

It is worth remarking again that the function f is defined differently for each kind of consecutive path. Given a function $first$ that returns the first node of the path defined by the reference to the previous node in a node in G_t, f is defined as:

- Earliest-arrival path: $f(x, y) = x.time - y.time$.
- Latest-departure path: $f(x, y) = first(x).time - first(y).time$
- Shortest path: $f(x, y) = x.length - y.length$
- Fastest path: $f(x, y) = (x.time - first(x).time) - (y.time - first(y).time)$

The library that has been developed, also contains aggregation functions. These functions iterate over the results and then return some value associated with the input. They are used to filter the results obtained by executing the consecutive paths procedure. They iterate over all the results received by the execution of these procedures, and choose the fastest, earliest, shortest, latest departure or latest arrival paths depending on the function we called. These functions are useful when the procedures are called more than once, for preventing returning non-optimal values.

Example 4 (Consecutive Paths Computation) Figure 10 shows a graph over which the shortest path between nodes A and B is computed with Algorithm 3. The function f will thus be $f(x, y) = x.length - y.length$. Figure 11 shows the transformed graph at the end of the execution of the algorithm.

The first node created in G_t is $(A, -\infty, 0)$ (the reference to the previous node is omitted, for clarity), which is added to the queue. Thus, $Q = [(A, -\infty, 0)]$ is the initial state of the queue. The node is picked up from the queue, and, since the edges outgoing from A in the graph of Figure 10 have intervals $[2, 4]$ and $[1, 2]$, taking $[2, 4]$, the nodes $v_{out} = (A, 2, 0)$ and $v_{in} = (C, 4, 1)$ are created in G_t. Then, v_{in} is picked up, and the edges outgoing from C have intervals $[5, 7]$, $[1, 3]$ and $[6, 8]$. Here,

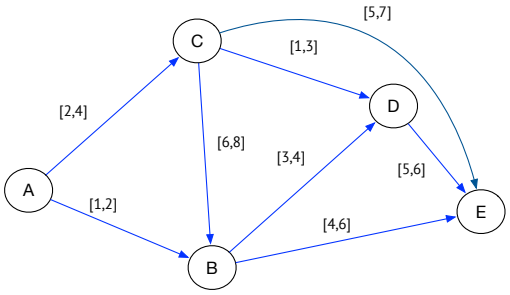

Fig. 10 An example for Algorithm 3.

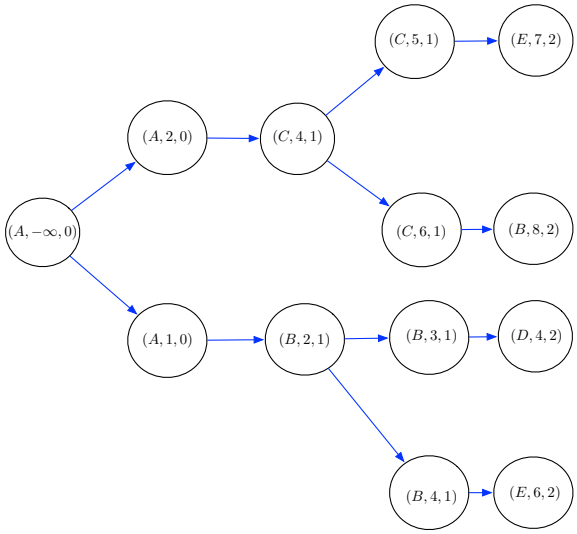

Fig. 11 Result of the execution of Algorithm 3.

$[1,3]$ cannot be expanded, since it would not yield a consecutive path. The new nodes v_{in} are created. From these nodes, and $(E,7,2)$ is added to the result set, and the new state of the queue is $Q = [(B,8,2),(B,2,1)]$. Since now a first solution is obtained, it is compared against $(B,8,2)$, and given that $f((B,8,2),(E,7,2)) = 2-2 = 0$, this path is discarded. Then, $(B,2,1)$ is expanded, and the process continues in the same way. Finally, the two paths are: $(A,1,0)- > (B,2,1)- > (B,4,1)- > (E,6,2)$ and $(A,2,0)- > (C,4,1)- > (C,5,1)- > (E,7,2)$ which leads to the shortest paths A,B,E and A,C,E.

5.4 Extending the system

At this point, the reader may be asking herself, whether or not the ideas exposed in this section can be generalized to other databases and query languages. Assume for example, that Janusgraph is the database that the user wants to use for storing the graph. The statements below allow representing the fact that a person A was a friend of B between 1999 and 2003, and between 2005 and 2015 (note that since `edge` is a reserved word in Gremlin, the edge variable below is ended with '_'). First, the schema is created, as follows:

```
edge_ = mgmt.makeEdgeLabel('edge_').
        multiplicity(SIMPLE).make()
livesin = mgmt.makeEdgeLabel('livesin').
          multiplicity(SIMPLE).make()
friend = mgmt.makeEdgeLabel('friend').
         multiplicity(SIMPLE).make()
object = mgmt.makeVertexLabel('Object').make()
Attribute = mgmt.makeVertexLabel('Attribute').
            make()
Value_= mgmt.makeVertexLabel('Value_').make()
title = mgmt.makePropertyKey('title').
        dataType(String.class).make();
val = mgmt.makePropertyKey('value').
      dataType(String.class).make();
int = mgmt.makePropertyKey('interval').
      dataType(String.class).
      cardinality(Cardinality.set).make();
friend = mgmt.makeEdgeLabel('friend').
         multiplicity(SIMPLE).make()
```

The edge labels are created first, followed by the vertex labels, which support the object, attribute and value nodes of the model. Finally, the property labels are created. The multiplicity 'simple' tells that only one edge type is allowed between the same two nodes. After declaring the schema, vertex and edges can be created as follows (attribute and value nodes are omitted for brevity):

```
o1=g.addV('object').property('title_','Person')
o2=g.addV('object').property('title_','Person')
f1=o1.addEdge('friend',o2,'interval',
              ['1999-2003','2005-2015'])
```

It can be seen that the creation of the graph is straightforward. Simple queries can also be easily generalized. Consider the query "*When was Pauline Boutlier friend of Cathy Van Bourne?*". This query is expressed in Gremlim as (the T-GQL query is omitted) follows, yielding the answer: $[1999-2003, 2005-2015]$.

```
g.V().hasLabel('Value').has('value','Pauline
Boutlier').in().in().outE('friend').
has('interval').as('f').inV().out().out().
has('value','Cathy Van Bourne').select('f').
by('interval').dedup()
```

However, generalizing the target language to address temporal paths (continuous and consecutive) is much more involved, since Gremlin must be non-trivially extended to be used as a target language for T-GQL. This can be inferred from the work by Byun at al. [9],

who show that the Gremlin's path management scheme must be modified, and new functions defined. This is left for future work, as explained in Section 8.

6 Evaluation

This section reports and discusses the experiments carried out in order to test the different algorithms described and implemented in this work. These experiments cover the two classes of path algorithms studied: continuous paths and consecutive paths. Since the implementation is a proof-of-concept, and Neo4j is not a database oriented and handling very large graphs, this evaluation is aimed at finding out the impact of the factors that influence the performance, rather than to measure performance itself. Future work will address performance issues through indexing schemes.

6.1 Description of the experiments

The goals of the experiments, and the experimental setup are detailed in this section, for each of the classes of algorithms tested.

6.1.1 Continuous paths algorithms

The goal of these experiments is to test how does the length of the paths and the size of the data set impact on the performance of the algorithm. Therefore, different tests are conducted, varying both variables. Typical continuous path queries are run over the social network temporal graph, asking for continuous paths of different lengths between two specific persons, the latter indicated by a property denoted id, generated during the population of the data set. For example, the query below asks for all the continuous paths of length 8 between the Person nodes with id 10 and 30. This query is run for different pairs of persons and different path lengths.

```
SELECT p
MATCH (n:Person), (m:Person),
       p = cPath((n)-[:Friend*8]-(m))
WHERE n[id] = 10 AND m[id] = 30
```

The same type of query was ran to test the pairwise continuous path algorithm:

```
SELECT p
MATCH (n:Person), (m:Person),
       p = pairCPath((n)-[:Friend*8]-(m))
WHERE n[id] = 10 AND m[id] = 30
```

6.1.2 Consecutive paths algorithms

The goal of these experiments is to evaluate how do the different paths behave for various graph sizes. The tests are run over real-world flights data sets, taking a subset of the airports in such data sets. The chosen airports are of very different sizes, to cover a wide range of connecting flights. The queries perform a consecutive path search for two specific airports using their IATA (International Air Transportation Association) code, a three-letter code that uniquely identifies an airport. The queries address the four kinds of consecutive path algorithms, and are of the following form:

```
SELECT path
MATCH (a1:Airport), (a2:Airport),
path = fastestPath((a1)-[:Flight*]->(a2))
WHERE a1.Code = 'BOS' and a2.Code = 'HOU'
```

```
SELECT path
MATCH (a1:Airport), (a2:Airport),
path = shortestPath((a1)-[:Flight*]->(a2))
WHERE a1.Code = 'BOS' and a2.Code = 'HOU'
```

```
SELECT path
MATCH (a1:Airport), (a2:Airport),
path = earliestPath((a1)-[:Flight*]->(a2))
WHERE a1.Code = 'BOS' and a2.Code = 'HOU'
```

```
SELECT path
MATCH (a1:Airport), (a2:Airport),
path = latestDeparturePath((a1)-[:Flight*]
       ->(a2))
WHERE a1.Code = 'BOS' and a2.Code = 'HOU'
```

6.1.3 Temporal model overhead

These experiments aim at evaluating the cost of introducing temporal support to a non-temporal system. That is, to measure the overhead in terms of memory and performance, produced by the structure described in Section 5. Two new social network graphs G_1 and G_2 are created: (a) a temporal one, called G_1, similar to the graph in Figure 2; and (b) a static graph G_2, derived from G_1, but keeping only the nodes and edges valid at a certain time (the current time), that is, a snapshot of G_1 at the instant 'Now'. Over these graphs, two queries are tested, both considering only the current instant.

- Query 1: *Compute all the paths of length 5 between every pair of persons.*
- Query 2: *Compute all the paths of length 5 starting from a given person.*

Both queries are run over the underlying Neo4j graph and over the static graph. For example, Query 2 over the underlying Neo4j graph reads:

```
MATCH p=(v1:Value)<--(a1:Attribute)<--
 (o1:Object{title:'Person'})-[e1:Friend]->
 (:Object{title:'Person'})-[e2:Friend]->
 (:Object{title:'Person'})-[e3:Friend]
 ->(:Object{title:'Person'})-[e4:Friend]->
 (:Object{title:'Person'})
 -[e5:Friend]->(o2:Object {title:'Person'})
WHERE v1.value ='Vivian Medhurst Jr.'
AND a1.title="Name" AND right(e1.interval
[size(e1.interval)-1],3)='Now'  AND
right(e2.interval[size(e2.interval)-1],3)
='Now' AND right(e3.interval[size
(e3.interval)-1],3)='Now' AND
right(e4.interval[size(e4.interval)-1],3)
='Now' AND right(e5.interval[size
(e5.interval)-1],3)='Now' AND
right(v1.interval[0],3)='Now'
RETURN p
```

The equivalent query over the static graph reads:

```
MATCH p=(o1:Object {title:'Person'})
 -[e1:Friend]->(:Object{title:'Person'})
 -[e2:Friend]->(:Object{title:'Person'})
 -[e3:Friend]->(:Object{title:'Person'})
 -[e4:Friend]->(:Object{title:'Person'})
 -[e5:Friend]->(o2:Object{title:'Person'})
WHERE o1.name ="Vivian Medhurst Jr."
RETURN p
```

6.2 Data sets and setup

This section reports the characteristics of the data sets used for evaluating the two kinds of algorithms. For continuous paths algorithms, synthetic data were generated, resembling the social network running example (Figure 2). For consecutive paths algorithms, real-world flight data were used.

All experiments were run under the same environment, a Neo4j 3.5.17 server, ran on Ubuntu 16.04 64-bits, with a 12-core CPU and 25 GB of RAM.

6.2.1 Continuous paths algorithms

A data set generator, based on the model described in Definition 4 and represented in Figure 2, populates the graph databases for these experiments. To generate data for the social network graph, the following parameters are considered:

- N = Number of Person nodes.
- F = Maximum number of Friend relationships per person.
- I = Maximum number of intervals per friendship.
- Number (C) and length (L) of the continuous paths.

First, the generator creates C continuous paths of length L and then, randomly generates the friendship relationships for the whole graph. The generator ensures a *minimum* of C continuous paths of length L. Once the continuous paths are created, the id of the persons involved in each continuous path are stored, to be used in the queries as the start and end of the paths of length L.

Three data sets were generated, with N = 1000, 10000 and 100000, and the other parameters are fixed, with values F = 5 and I = 2. For each data set, at least 3 paths (i.e., $C = 3$) of each of the following lengths (L) were generated: 4, 6, 8, 10 and 12. Table 1 details the number of nodes, edges and sizes of the data sets. Indices were created on the Object, Value and Attribute nodes for the id property.

N	Nodes	Edges	Size
1000	3021	6833	747.95 MB
10000	30021	67676	776.02 MB
100000	300021	677278	1.06 GB

Table 1 Continuous paths experiments: Characteristics of each social network data set.

The execution of a query for a specific N and L is carried out C times varying the ids of the start and end nodes of the path, to account for different number of paths of length L that may exist in a graph, and for the different starting and ending nodes.

6.2.2 Consecutive paths algorithms

Consecutive path algorithms were tested using a real-world flight database, the Flight Delays and Cancellations for US flights in 2015[8], using the original departure and arrival times for the flights. Five data sets were generated from the former ones, filtering the flights with different time intervals. The selected periods for the data sets were the first week, first month, first three months, first half year, and the entire year. The number of flights and airports are shown in Table 2. The following airports were chosen:

1. ATL - Atlanta International Airport, Atlanta, GA.
2. CLD - Mc Clellan-Palomar Airport, Carlsbad, CA.
3. BOS - Logan International Airport, Boston, MA.
4. HOU - William P. Hobby Airport, Houston, Texas
5. SBN - South Bend Regional Airport, S. Bend, IN.
6. ISP - Long Island Mac Arthur Airport, Islip, NY.

The selected routes between these airports were:

[8] https://www.kaggle.com/usdot/flight-delays?select=flights.csv

data set	Airports	Flights	Size
1 week	312	109911	1.92MB
1 month	312	469968	22.53 MB
3 months	315	1403471	64.52 MB
6 months	322	2889512	131.38 MB
1 year	629	5819079	413.23 MB

Table 2 Consecutive paths experiments: Number of airports, flights and sizes of each data set.

data set	V_{out}	V_{in}	Total
1 week	71455	84216	155661
1 month	308656	366301	674957
3 months	920257	1095713	2015970
6 months	1891583	2254938	4146521
1 year	3828264	4549494	8377758

Table 3 Total number of nodes in each data set.

1. ATL to CLD (A large airport to a small one)
2. BOS to HOU (A medium-size airport to a medium-size one)
3. ATL to AUS (A large airport to a medium one)
4. SBN to ISP (A small airport to a small one)

Routes between two large airports were not chosen because usually there are direct flights between them, meaning that a path of length 1 normally exists, and therefore the results would not be representative. The number of incoming and outgoing flights are listed in Table 8 in Appendix A. Note that for CLD airport, the number of flights stops growing at the 6 months as the airport closes. This airport was chosen since it challenges the latest departure path algorithm, as it will try to search for the latest departure path going to the paths with the latest departure time, although the arrivals are all in the first half of the year.

6.2.3 Temporal model overhead

It was mentioned that for assessing the overhead introduced by the temporal graph model, two new social network graphs G_1 and G_2 are created. The former has 52,000 nodes, 10,000 of them labeled as Object nodes of type Person, 20,000 labeled as Attribute nodes, and 22,000 nodes labeled as Value nodes. Every Person node has two attribute nodes: name and identifier, whereas every attribute node has its corresponding value node. G_1 has 43,097 Friend relationships.

From G_1, a static graph G_2 is obtained, keeping only the nodes and edges valid at the current time, with no reference to time at all, i.e., G_2 is a snapshot of G_1 at time instant 'Now'. The steps to take this snapshot are (assume a copy of G_1, denoted G_2 is created):

1. Select nodes and edges whose interval end with 'Now'.
2. Two properties, name and identifier, are added to the Object nodes. Thus, so far, G_1 has zero or one value node per attribute node, because each different value of the title property is collapsed as a property name of the object node, and the values of these properties correspond to the associated value node.
3. All attribute and value nodes are deleted from G_2.
4. All time intervals are deleted from G_2.

The resulting graph G_2 has 8,000 Object nodes and 19,370 Friend relationships.

6.3 Results

This section reports the results of the experiments presented above. The algorithms' execution times depend on a number of factors, like, for example, the number of continuous paths of a certain length that the algorithm finds, which may vary for different pairs of starting and ending nodes. Therefore, to ensure a fair comparison, the following average definition is used.

$$T = \frac{1}{n}\sum_{i=1}^{n}\frac{t_n}{c_n} = \frac{1}{n}(\frac{t_1}{c_1} + \ldots + \frac{t_n}{c_n})$$

In the expression above, n is the number of different pairs of nodes (start and end of a continuous path) for which the query was run, t the execution time and c the number of paths found for each pair of nodes. For example, for $C = 3$, a minimum of three continuous paths of length L are generated between three pairs of nodes, but more could be found.

Node pair	Paths found	Execution time
$A_1 \to A_2$	3	12 s
$A_3 \to A_4$	2	6 s
$A_5 \to A_6$	9	45 s

The weighted average T is computed as:

$$T = \frac{1}{n}(\frac{t_1}{c_1} + \frac{t_2}{c_2} + \frac{t_n}{c_n}) = \frac{1}{3}(\frac{12}{3} + \frac{6}{2} + \frac{45}{9}) = 4$$

For consecutive paths, the usual definition of average is used, running the algorithms three times for each path and data set.

Figure 12 displays the execution times for the continuous path and pairwise continuous path algorithms. The x-axis represents the length of the continuous paths in the queries. Figure 13 displays the execution times with respect to the number of nodes visited by the continuous path algorithm, for $N = 100,000$ and $L = 12$. In this case, the execution time is the simple average computed dividing the execution time by the number of paths found for each pair of person nodes.

Figures 14 and 15 display the results for the tests addressing latest departure, fastest, earliest, and shortest paths algorithms. Execution times are represented on the y-axis, and the number of flights on the x-axis. Table 3 shows the number of V_{out} and V_{in} nodes of the complete transformed graph, for each data set. Tables 4 through 7 show, for each route tested, the average time and the number of paths in the result, for each time partition of the data set.

Path	data set	Latest Departure Path	
		Avg. Time (ms)	# Results
ATL → CLD	1 week	267	1
	1 month	1318.33	1
	3 months	4098	1
	6 months	15622280.67	1
	1 year	129165589.33	1
BOS → HOU	1 week	231	1
	1 month	1224.33	3
	3 months	3952.33	1
	6 months	12072	1
	1 year	33875.33	1
ATL → AUS	1 week	97.33	1
	1 month	1807	2
	3 months	7462.33	1
	6 months	34883	1
	1 year	118174.67	1
SBN → ISP	1 week	257	1
	1 month	1263.67	9
	3 months	3735.33	3
	6 months	8829.67	3
	1 year	18760.33	74

Table 4 Average time and number of results for the latest departure path algorithm.

6.3.1 Temporal model overhead

Figure 16 shows the space required by the two graphs, G_1 and G_2, the temporal and static ones, respectively. It can be seen that, although there is a large overhead produced by the structural information, most of the space required by G_1 is used to store actual temporal information, that is, the history of the graph. This space overhead does not impact in the same way on the performance results shown in Figure 17. This figure shows the results of executing Queries 1 and 2 indicated in Section 6.1.3. In both queries, the performance overhead lies between 25 and 30%, which appears to be reasonable given the space overhead introduced by the structure needed to keep the historical information.

6.4 Discussion of Results

A discussion of the results reported in the previous section is presented next.

Path	data set	Fastest Path	
		Avg. Time (ms)	# Results
ATL → CLD	1 week	6755	3
	1 month	140522.33	3
	3 months	1427239.33	3
	6 months	8172404	8
	1 year	29744579	8
BOS → HOU	1 week	1969.33	7
	1 month	51980.67	31
	3 months	536338	31
	6 months	2123572.67	2
	1 year	8694658.33	11
ATL → AUS	1 week	973	3
	1 month	17640.33	27
	3 months	237272.33	45
	6 months	671548	21
	1 year	2938933	4
SBN → ISP	1 week	3925.33	1
	1 month	72191.33	2
	3 months	560382	4
	6 months	3628807	21
	1 year	13622908.67	1

Table 5 Average time and number of results for the fastest path algorithm.

Path	data set	Earliest Path	
		Avg. Time (ms)	# Results
ATL → CLD	1 week	412	1
	1 month	1995.33	1
	3 months	7349	1
	6 months	18505	1
	1 year	36813.67	1
BOS → HOU	1 week	360	1
	1 month	1783.33	1
	3 months	5699.67	1
	6 months	14219.66	1
	1 year	33812	1
ATL → AUS	1 week	98.33	1
	1 month	414.33	1
	3 months	1411.33	1
	6 months	2758.33	1
	1 year	6391.67	1
SBN → ISP	1 week	1992.67	9
	1 month	10507	9
	3 months	36670.33	9
	6 months	102361.67	9
	1 year	238015.67	9

Table 6 Average time and number of results for the earliest path algorithm.

6.4.1 Continuous Paths

The left-hand side of Figure 12 shows the execution time for each data set size, and different continuous path lengths. For $N = 10000$ and 100000, the execution times increase as the path length increases, starting with values around 50 ms for $L = 4$ and growing up to 733 ms and 3279 ms, respectively for L = 12. On the other hand, for $N = 1000$, execution times remain low, and, starting with an execution time of 30 ms, de-

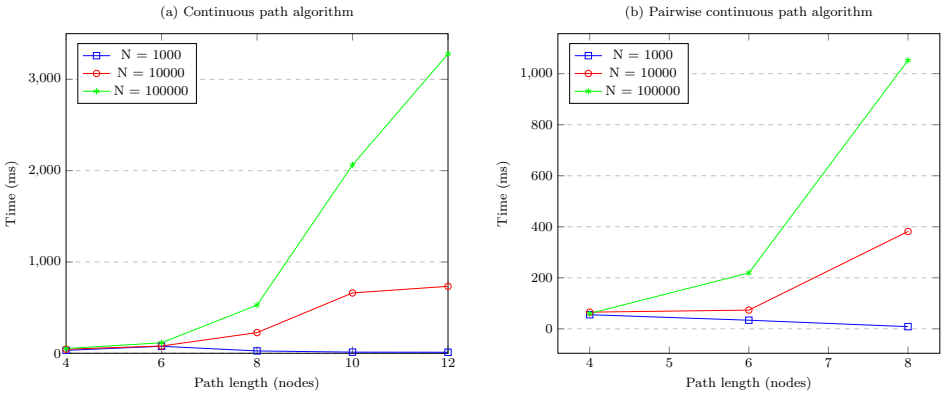

Fig. 12 (a) Execution time vs. Path length for continuous path algorithm; (b) Execution time vs. Path length for pairwise continuous path algorithm.

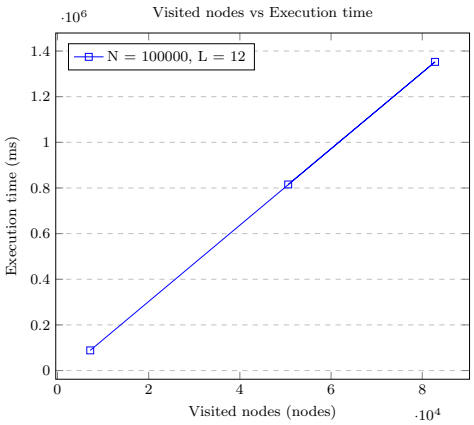

Fig. 13 Visited nodes vs. Execution time for continuous path algorithm on paths of $L = 12$.

creases for longer paths, without exceeding 80 ms in any case. It can also be seen that, for $N = 100000$, execution times grow faster than for $N = 10000$. Figure 13 shows that the execution time is linear with respect to the number of nodes visited by the algorithm. Results for the pairwise continuous paths are depicted on the right-hand side of Figure 12. Relative to each other, results for the three data sets are similar to the ones obtained for continuous paths: increasing the length of the path searched implies higher execution times. However, it can be seen that execution times are lower in this case.

6.4.2 Consecutive Paths

Figures 14 and 15 display the results for the tests addressing latest departure, fastest, earliest, and shortest paths algorithms. All figures show a lineal behaviour in most of the cases. The y-axis is displayed in logarithmic scale, since the difference between the running times of the algorithms is very large, depending on the paths.

Path	data set	Shortest Path	
		Avg. Time (ms)	# Results
ATL → CLD	1 week	4031.67	1969
	1 month	91449.67	39034
	3 months	1060364.67	342124
	6 months	4643210	391462
	1 year	Out of Memory	
BOS → HOU	1 week	10.67	20
	1 month	83.67	89
	3 months	82	253
	6 months	342.67	506
	1 year	469.33	926
ATL → AUS	1 week	32.33	58
	1 month	99	252
	3 months	273	775
	6 months	585.67	1667
	1 year	1252.67	3154
SBN → ISP	1 week	8066	2783
	1 month	270057.33	66464
	3 months	3459141.33	699214
	6 months	Out of Memory	
	1 year	Out of Memory	

Table 7 Average time and number of results for the shortest path algorithm.

As expected, the execution time of the algorithm grows as the number of flights grows.

For the *latest departure path* (Figure 14 (a) and Table 4), tests show a rather linear behaviour except the one from ATL to CLD. This is because all the arrivals to the airport are in the first half of the year, so it takes a long time to prune the graph to find a path between those airports. This is why the time grows exponentially and then continues linearly, reflected in the fact that the algorithm runs in 4098msec for the 3-months-data set, and 15622280.67msec for the 6-months one, that is, a growth of about 3800 times. For the other airports, this ratio is between 3 and 5. However, note that this is a very particular case.

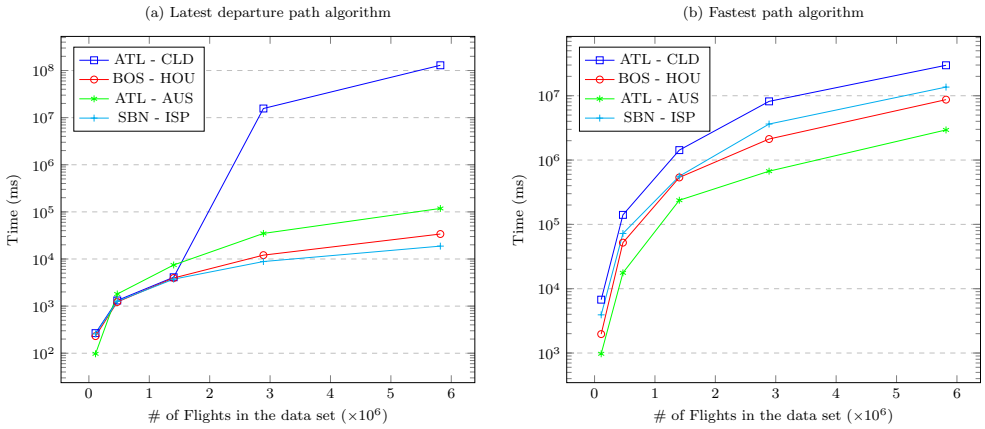

Fig. 14 (a) Execution time for each pair of airports for the latest departure path algorithm; (b) Execution time for each pair of airports for the fastest path algorithm.

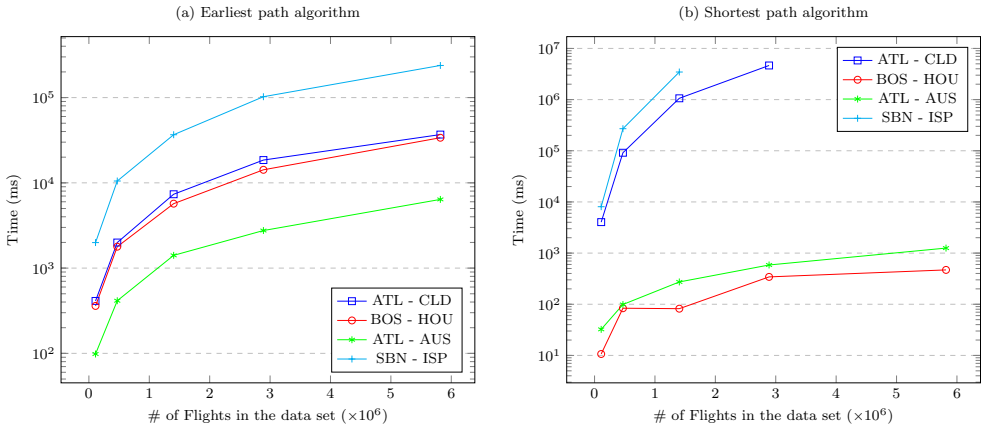

Fig. 15 (a) Execution time for each pair of airports for the earliest path algorithm; (b) Execution time for each pair of airports for the shortest path algorithm.

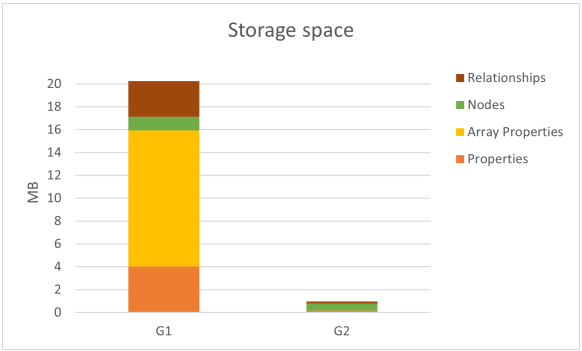

Fig. 16 Performance results for the snapshot queries.

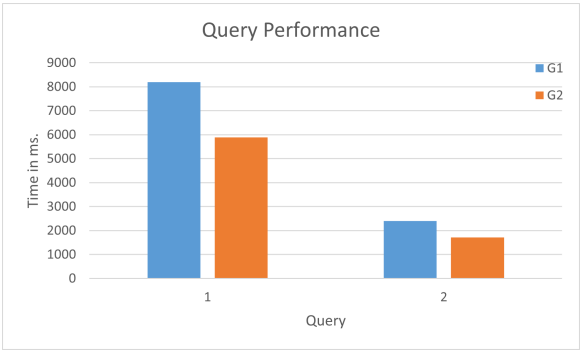

Fig. 17 Space overhead results.

In the case of the shortest path algorithm (Figure 15(b) and Table 7), for the largest data set, and routes including one small airport, the algorithm ran out of memory, due to the large number of results obtained. This was caused by the number of paths that are stored in the memory. Another particular situation occurs when a path starts in the beginning of the year, and ends at the end of the year.

For all algorithms, with the exception of the cases of shortest and latest departure paths mentioned above, the behaviour of the algorithms is rather linear, and, in many cases, not dependent on the routes (it can be seen that, in general, the curves are quite close to each

other). The fastest path algorithm (Figure 14(b) and Table 5) is the one with the lowest performance (except from the particular cases mentioned above). For example, for the largest data set, for the paths from BOS to HOU and ATL to AUS, the average execution times were 8,694,658msec and 2,938,933msec respectively. On the other hand, for the earliest path (Figure 15 and Table 6), these times were 33,812ms and 6,391.67ms, for the shortest path 469.33ms and 1252.67ms, and for the latest departure 33,875ms and 118,174ms, respectively. However, as the sizes of the data sets decrease, the execution times also decrease in a significant way. For example, in the case of the fastest path algorithm, for the 1-month data set, for the paths from BOS to HOU and ATL to AUS, the average execution times were 51,980msec and 17,640msec respectively.

The intuition is that the results reported may be caused by the nature of the paths. For the earliest departure path algorithm, execution time depends on the time of the last node in the path; for the latest departure path algorithm, the execution time strongly depends on the time of the first node of the path, and the shortest path on the length of the path. For example, in the latest departure path algorithm, once a path reaches a node that is part of a possible latest departure path, no better path can be reached that contains that node, because the time of the first node cannot change. On the other hand, in the fastest path algorithm, a fastest path could be found, depending on the first and last nodes. The shortest path algorithm explores the same node many times, increasing the execution time.

7 Indexing

This section studies how the performance of the queries presented in this paper can be enhanced through the use of indexing strategies. First, an indexing scheme is proposed. Then, the impact of updates is studied. Finally, a sketch of how the index could be used when a query is submitted to the system, is given. It is worth mentioning that these strategies are ongoing research work, and they are included here to highlight the issues that need to be addressed in future work.

7.1 Temporal Indices

Typically, indices on property graph databases are defined over properties (in Neo4j, only properties in nodes can be indexed). The research community has proposed indexing schemes for paths (in particular, shortest paths) in non-temporal (static) graphs. For example, Kusu and Hatano [39] index the shortest path between every two nodes in a subgraph of a given one. Hassan et al. [30] propose a method to index paths usually mentioned in queries (these paths are called *recurrent*). In another proposal, Pokorny et al. [42] index graph patterns in Neo4j, using a structure stored in the same database as the graph.

To enhance the performance of the queries discussed in this paper, the three kinds of paths studied (i.e., continuous, pairwise continuous, and consecutive) can be indexed. Since these paths are temporal, indexing must also be applied along the time dimension, for example, to quickly find the continuous paths within a given interval. A sketch of a possible solution for this problem is discussed next. The idea is based on the proposal in [42], which can be combined with typical methods for time indexing [18], to produce different kinds of indices, one for each type of path addressed in the paper. This way, as proposed in [43]), there is an entry in the index for each path in a given interval. For continuous paths, for example, the index is implemented as a collection of Neo4j nodes containing, as properties, the start and end nodes, the nodes in the path, and the time interval of the continuous path. Figure 18 shows an example over the social network graph. Here, all relationships other than Friend are indicated in dashed lines, for clarity. There is an index for the continuous paths over this relationship. Two index entry nodes are shown (in black), for paths of length 2 and 4. The former starts at the Person node with id=22, and ends at node with id=20, with interval $[2010-2017]$. The latter starts at the Person node with id=22, and ends at node with id=20, with interval $[2015-2017]$. There is a pointer from the index entry node, to each of the nodes in the indexed paths. Of course, many issues must be considered in this scheme, and are open research problems. For example, indexing all possible paths would result in huge index volumes, where probably a large portion of the index would remain unused. Therefore, incremental indexing based on the prediction of the queries mentioning a certain path can be applied. Another issue refers to updating the index as changes on the underlying graph occur. This is discussed below.

7.2 Updating the index

The updates presented in Section 3.4, in general, impact on the state of the indices discussed above. For example, if an edge is added to the graph, it may produce an update in one or more index entries. The paths related to the inserted edge must be recomputed and the index be updated if the new edge, for instance, produces a new continuous path. Note that if a node is added to the graph, this has no effect on the path index until an

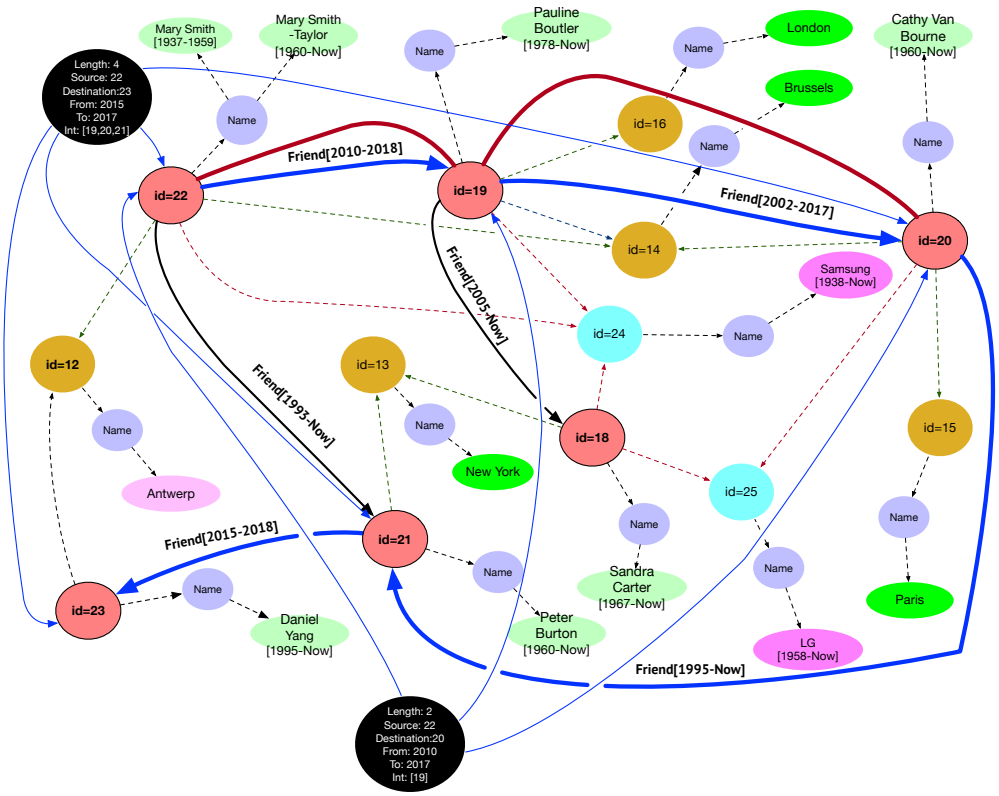

Fig. 18 Indexing paths in a temporal graph.

edge connects the new node with the existing graph. The impact over the continuous path, of the insertion and update of an edge, is studied next.

Insertion of an edge Given a graph G, assume that a new edge e_{new} from v_1 to v_2 is added to G, with time interval $[t_d - Now]$, where t_d is the current timestamp. This change produces new paths if e_{new} is connected with paths valid at the current instant, Now. In this case, the existing paths ending at v_1 and starting at v_2 are extended with e_{new}. The time interval for the new paths would be $[t_d - Now]$, and they must be added to the index, if it exists. Algorithm 4 describes the procedure.

Updating an edge Assume there is an edge e_{old}, whose ending time is $t_{old} \neq Now$. The new edge interval $e_{new}.interval$ would be $\{e_{old}.interval, [t_d - Now]\}$. This change may produce new paths if e_{old} is connected to paths valid at the Now time instant. In this case, the paths ending at v_1 and starting at v_2 will be extended with e_{new}. The time interval for the new paths is $[t_d - Now]$. Algorithm 5 describes the procedure.

Deleting an edge This operation is only possible for an edge e_{old} whose ending time is $t_f = Now$. The new ending time of edge, e_{new}, becomes the current timestamp t_d. The algorithm looks for every path P valid at Now, such that P includes e_{old}. The ending time of all

Algorithm 4 Index Update: Edge Insertion

Input: A temporal graph G, a temporal Index I, a new edge e_{new} with starting and ending nodes v_1 and v_2, respectively.

Output: An updated Index I'.

Initialize a list Out with all indexed cPaths whose Source node is $v2$, valid at Now time instant.
Initialize a list In with all indexed cPaths whose Destination nodes is $v1$, valid at Now time instant.
Initialize an empty list P_{temp}
for all $path_i \in Out$ **do**
  $path = \{v_1 + e_{new} + path_i\}$
  $path.interval = [t_d - Now]$
  Add $path$ to P_{temp}
end for
Add the path $v_1 + e_{new} + v_2$ to P_{temp} {This is a path of length 1.}
Copy P_{temp} to P_{final}
for all $path_i \in P_{temp}$ **do**
  for all $path_j \in In$ **do**
    $path = \{path_j + path_i\}$
    Add $path$ to P_{final}
    $path.interval = [t_d - Now]$
  end for
end for
For every path in P_{final} create a new temporal index node.

Algorithm 5 Index Update: Edge Update

Input: A temporal graph G, a temporal Index I, an edge e_{old} with starting and ending nodes v_1 and v_2, respectively, and the updated edge e_{new}.
Output: An updated Index I'.
Initialize a list Out with all indexed cPaths starting at $v2$ valid at Now time instant.
Initialize a list In with all indexed cPaths ending at $v1$ valid at Now time instant.
Initialize an empty list P_{temp}
for all $path_i \in Out$ **do**
  $path = \{v_1 + e_{new} + path_i\}$
  $path.interval = [t_d - Now]$
  Add $path$ to P_{temp}
end for
Add the path v_1+e_{new}+v_2 to P_{temp}. {This is a path of length 1.}
Copy P_{temp} to P_{final}
for all $path_i \in P_{temp}$ **do**
  for all $path_j \in In$ **do**
    $path = \{path_j + path_i\}$
    $path.interval = [t_d - Now]$
    Add $path$ to P_{final}
  end for
end for
For every path in P_{final} create a temporal index node

Algorithm 6 Index Update: Edge Delete

Input: A temporal graph G, a temporal Index I, an edge e_{old} where the interval ends at Now, an edge e_{new} where the interval ends at the current time t_d, and its starting and ending nodes v_1 and v_2.
Output: An updated Index I'.
Initialize a list $Current$ with all indexed cPaths valid at Now time, that include v_1+e_{old}+v_2
for all $path_i \in Current$ **do**
  $p_{interval} = path_i.interval$
  $t_s = p_{interval}.startTime$
  $path_i.interval = [t_s - t_d]$
end for
Update the index nodes associated with $Current$

those paths stored in the index, must be changed to t_d. Algorithm 6 shows the process.

7.3 Using the index

To conclude this discussion, this section sketches how the indices above can be used to enhance performance. As usual, the idea is that if a T-GQL query asks for a certain path, if possible, it will be redirected to an index. If, for example, the query mentions a source value for the path, the object identifier associated with such value must be looked for. Any path starting from that object node, will have a relationship labeled start connecting it to the index node representing that path. There can be as many index nodes connected to the object node, as paths starting from it in the graph. Thus, the answer to the query will be the index nodes that match the parameters. An example is shown next. The query below asks for the continuous paths of length 2 starting from the node corresponding to Mary Smith-Taylor.

```
SELECT p2.interval
MATCH (p1:Person),(p2:Person),
paths = cPath((p1)-[:Friend*2]->(p2))
WHERE p1.Name = "Mary Smith-Taylor"
```

The query processor will find out that there is an index for the relationship, like the one in Figure 18, and will translate the query into Cypher as follows:

```
MATCH (v1:Value)<--()<--(o1:Object)
      <-[:start]-(i:Index)
WHERE v1.value = "Mary Smith-Taylor"
      AND o1.id=i.source AND i.length=2
RETURN [i.from,i.to]
```

8 Conclusions and Future Research Directions

The first part of this section summarizes the paper and its results. In light of these results, the second part of the section motivates and discusses future work.

8.1 Paper summary

This work introduces a temporal property graph data model, and an associated high-level temporal query language, denoted T-GQL, which supports two kinds of temporal paths semantics: continuous paths (and the particular case of pairwise continuous) and consecutive paths. As relevant real-world application examples, those semantics capture the dynamics of social network evolution, and of travel scheduling, respectively. Algorithms for path computation for both semantics are devised and implemented. Finally, experiments are carried out, and the results reported and discussed.

The experiments address the tree kinds of paths mentioned above. For the continuous and pairwise continuous paths, the synthetic data sets simulating a social network are produced, with sizes up to 700,000 edges and 300,000 nodes. For consecutive paths, real-world flight data are used, with sizes up to 6,000,000 flights. Since the database is not optimized, the experiments are aimed at showing the plausibility of the approach, and highlighting the main issues that need to be addressed in future work. The results show execution times below 3.5 seconds for temporal paths up to a length of 12. For the consecutive path algorithms

times are in the (1,60)-seconds range for the largest data set, and below 1 second for the shortest-path algorithm, except for particular cases in the data sets, which affect the algorithms. All in all, the results suggest that the proposal is a plausible step towards temporal graph databases.

8.2 Open research directions

Although the current version of the T-GQL language has powerful features, it can be extended in many ways. Just as an example, the WHEN clause could be improved so it can support a path function call (even more than one such clauses can be supported). Further, the MATCH clause could be enhanced to support more than one path, as in the current version.

Performance is a key issue, particularly in temporal databases. Indexing is crucial to achieve an acceptable performance for (temporal) path queries. The ideas discussed in Section 7 for continuous paths, which borrow from existing research on indexing both, temporal databases and paths in graphs, must further investigated and extended to other kinds of paths.

Finally, although in this paper T-GQL is implemented over Neo4j, for larger graphs other options need to be investigated, as sketched in Section 5.4. This also requires a generalization of the proposal, allowing target languages other than Cypher. The goal is, for example, to allow using Janusgraph as the underlying database, and Gremlin as target language. This is not a trivial task, however, as shown in [9], where the authors not only extend Gremlin with temporal functions to support the computation of paths similar to the consecutive paths studied in this paper, but also define their own path management scheme, adapting the one used by Gremlin.

Acknowledgements Alejandro Vaisman and Valeria Soliani were partially supported by Project PICT 2017-1054, from the Argentinian Scientific Agency.

Conflict of interest

The authors declare no conflict of interest.

References

1. T. Amagasa, M. Yoshikawa, and S. Uemura. A temporal data model for XML documents. In *DEXA*, pages 334–344, 2000.
2. R. Angles. A Comparison of Current Graph Database Models. In *Proceedings of ICDE Workshops*, pages 171–177, Arlington, VA, USA, 2012.
3. R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.
4. Renzo Angles. 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*, volume 2100 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.
5. Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.
6. Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. RDF and property graphs interoperability: Status and issues. In Aidan Hogan and Tova Milo, editors, *Proceedings of the 13th Alberto Mendelzon International Workshop on Foundations of Data Management, Asunción, Paraguay, June 3-7, 2019*, volume 2369 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019.
7. Andrey Balmin, Thanos Papadimitriou, and Yannis Papakonstantinou. Hypothetical queries in an OLAP environment. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 220–231. Morgan Kaufmann, 2000.
8. Jaewook Byun. Enabling time-centric computation for efficient temporal graph traversals from multiple sources. *IEEE Trans. Knowl. Data Eng., Accepted for publication*, 2020.
9. Jaewook Byun, Sungpil Woo, and Daeyoung Kim. ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Trans. Knowl. Data Eng.*, 32(3):424–437, 2020.
10. Alexander Campos, Jorge Mozzino, and Alejandro A. Vaisman. Towards temporal graph databases. In Reinhard Pichler and Altigran Soares da Silva, editors, *Proceedings of the 10th Alberto Mendelzon International Workshop on Foundations of Data Management, Panama City, Panama, May 8-10, 2016*, volume 1644 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2016.
11. C. Cattuto, A. Panisson, and M. Quaggiotto. Representing time dependent graphs in Neo4j. `https://github.com/SocioPatterns/neo4j-dynagraph/wiki/Representing-time-dependent-graphs-in-Neo4j`, 2013.
12. Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch. Time-varying social networks in a graph database: a neo4j use case. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-loated with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, page 11, 2013.
13. C.X. Chen and C. Zaniolo. Universal temporal extesions for database languages. In *IEEE/ICDE*, Sydney, Australia, 1999.
14. S. Chien, V. Tsotras, and C. Zaniolo. Efficient management of multiversion documents by object referencing. In *VLDB*, pages 291–300, Rome, Italy, 2001.
15. James Clifford, Curtis E. Dyreson, Tomás Isakowitz, Christian S. Jensen, and Richard T. Snodgrass. On the semantics of "now" in databases. *ACM Trans. Database Syst.*, 22(2):171–214, 1997.
16. Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: an integrated API for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, page 2, 2016.

17. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
18. Ramez Elmasri, Yeong-Joon Kim, and Gene T. J. Wuu. Efficient implementation techniques for the time index. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 102–111. IEEE Computer Society, 1991.
19. Opher Etzion, Sushil Jajodia, and Suryanarayana M. Sripada, editors. *Temporal Databases: Research and Practice. (the book grow out of a Dagstuhl Seminar, June 23-27, 1997)*, volume 1399 of *Lecture Notes in Computer Science*. Springer, 1998.
20. Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language cypher. *CoRR*, abs/1802.09984, 2018.
21. Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018.
22. Matteo Golfarelli and Stefano Rizzi. What-if simulation modeling in business intelligence. *IJDWM*, 5(4):24–43, 2009.
23. Fabio Grandi. T-SPARQL: A tsql2-like temporal query language for RDF. In Mirjana Ivanovic, Bernhard Thalheim, Barbara Catania, and Zoran Budimac, editors, *Local Proceedings of the Fourteenth East-European Conference on Advances in Databases and Information Systems, Novi Sad, Serbia, September 20-24, 2010*, volume 639 of *CEUR Workshop Proceedings*, pages 21–30. CEUR-WS.org, 2010.
24. Alastair Green. Gql - initiating an industry standard property graph query language, 2018.
25. Claudio Gutiérrez, Carlos A. Hurtado, and Alejandro A. Vaisman. Temporal RDF. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *The Semantic Web: Research and Applications, Second European Semantic Web Conference, ESWC 2005, Heraklion, Crete, Greece, May 29 - June 1, 2005, Proceedings*, volume 3532 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.
26. Claudio Gutiérrez, Carlos A. Hurtado, and Alejandro A. Vaisman. Introducing time into RDF. *IEEE Trans. Knowl. Data Eng.*, 19(2):207–218, 2007.
27. W. Han, Y. Miao, K.Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A Graph Engine por Temporal Graph Analysis. In *Eurosys*, pages 1–14, 2014.
28. O. Hartig. Reconciliation of RDF* and property graphs. *CoRR*, abs/1409.3288, 2014.
29. O. Hartig. Position statement: The rdf* and sparql* approach to annotate statements in rdf and to reconcile rdf and property graphs, 2019.
30. Mohamed S. Hassan, Walid G. Aref, and Ahmed M. Aly. Graph indexing for shortest-path finding over dynamic sub-graphs. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1183–1197. ACM, 2016.
31. Huahai He and AmbujK. Singh. Query language and access methods for graph databases. In *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 125–160. Springer US, 2010.
32. Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *CoRR*, abs/1401.1919, 2014.
33. Wenyu Huo and Vassilis J. Tsotras. 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*, pages 38:1–38:4, 2014.
34. Theodore Johnson, Yaron Kanza, Laks V. S. Lakshmanan, and Vladislav Shkapenyuk. Nepal: a path query language for communication networks. In Akhil Arora, Shourya Roy, and Sameep Mehta, editors, *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics, NDA@SIGMOD 2016, San Francisco, California, USA, July 1, 2016*, pages 6:1–6:8. ACM, 2016.
35. U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. *CoRR*, arxiv:1207.5777, 2012.
36. U. Khurana and A. Deshpande. HiNGE: Enabling Temporal Analytics at Scale. In *Proceedings of SIGMOD*, NY, USA, 2013.
37. V. Kostakos. Temporal graphs. *CoRR*, arxiv:0807.2357, 2008.
38. Krishna G. Kulkarni and Jan-Eike Michels. Temporal features in SQL: 2011. *SIGMOD Rec.*, 41(3):34–43, 2012.
39. Kazuma Kusu and Kenji Hatano. Recurrent path index for efficient graph traversal. In *2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019*, pages 6107–6109. IEEE, 2019.
40. L.Lazarevic. Keeping track of graph changes using temporal versioning. `https://medium.com/neo4j/keeping-track-of-graph-changes-using-temporal-versioning-3b0f854536fa`, 2019.
41. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.
42. Jaroslav Pokorný, Michal Valenta, and Martin Troup. Indexing patterns in graph databases. In Jorge Bernardino and Christoph Quix, editors, *Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, Porto, Portugal, July 26-28, 2018*, pages 313–321. SciTePress, 2018.
43. F. Rizzolo and A. Vaisman. Temporal XML: Modeling, indexing, and query processing. *VLDB Journal*, 1179–1212(5):39–65, 2008.
44. I. Robinson, J. Webber, and Emil Eifrém. *Graph Databases*. O'Reilly Media, 2013.
45. Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In James Cheney and Thomas Neumann, editors, *Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015*, pages 1–10. ACM, 2015.
46. Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Trans. Knowl. Data Eng.*, 31(1):181–194, 2019.
47. Richard T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.
48. A. Tansel, J. Clifford, and S. Gadia (eds.). *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings, 1993.

49. Jonas Tappolet and Abraham Bernstein. Applied temporal RDF: efficient temporal querying of RDF data with SPARQL. In *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings*, volume 5554 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2009.
50. Paolo Terenziani and Richard T. Snodgrass. Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Trans. Knowl. Data Eng.*, 16(5):540–551, 2004.
51. Harsh Thakkar, Renzo Angles, Dominik Tomaszuk, and Jens Lehmann. Direct mappings between RDF and property graph databases. *CoRR*, abs/1912.02127, 2019.
52. Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.
53. Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. Efficient algorithms for temporal path computation. *IEEE Trans. Knowl. Data Eng.*, 28(11):2927–2942, 2016.
54. Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. Core decomposition in large temporal graphs. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 649–658, 2015.
55. Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 145–156, 2016.
56. Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John C. S. Lui. Diversified temporal subgraph pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1965–1974, 2016.

A Appendix A

Characteristics of the data sets

Airport	data set	Departing Flights	Arriving Flights
ATL	1 week	6707	6678
	1 month	29512	29492
	3 months	89632	89633
	6 months	186135	186180
	1 year	346836	346904
CLD	1 week	44	44
	1 month	204	204
	3 months	601	601
	6 months	641	640
	1 year	641	640
BOS	1 week	1943	1953
	1 month	8837	8841
	3 months	27188	27204
	6 months	57973	57996
	1 year	107847	107851
HOU	1 week	1105	1106
	1 month	4650	4651
	3 months	13628	13628
	6 months	27972	27972
	1 year	52042	52041
AUS	1 week	796	797
	1 month	3376	3372
	3 months	10182	10186
	6 months	21941	21950
	1 year	42067	42078
SBN	1 week	79	80
	1 month	384	386
	3 months	1246	1248
	6 months	2452	2455
	1 year	4454	4452
ISP	1 week	88	89
	1 month	377	378
	3 months	1162	1163
	6 months	2462	2463
	1 year	4392	4392

Table 8 Number of incoming and outgoing flights for each airport.

B Appendix B

Summary of main concepts

Symbol	Name	Meaning
$G(N_o, N_a, N_v, E)$	Temporal property graph	A graph where nodes and edges are labelled with their interval of validity, and the history of node properties and values is kept. The properties of the edges is are static.
N_o, N_a, N_v	Sets of object, attribute, value nodes	Object nodes in N_o represent entities, attribute nodes in N_a represent time-varying properties of an object node, and value nodes in N_v represent time-varying property values.
$n.interval, e.interval$	Time interval associated with a node n or an edge e.	
$CP = (n_1, \ldots, n_k, r, T)$	Continuous path between $n_1$and n_k, with interval T and edge type r.	$(n_1, \ldots, n_k, r, T)$ of k nodes and an interval T Sequence of consecutive edges between $n_1$and n_k valid during an interval that is the intersection of all the intervals in the path.
$PCP = (n_1, \ldots, n_k, r)$	Pairwise continuous path between n_1 and n_k and edge type r.	Sequence of edges such that there is an intersection in the interval of two consecutive ones.
$P_c = (n_1, n_2, r, [t_1, t_2])$	Consecutive path	A sequence of edges such that the end time of an edge interval is less or equal than the starting time of the immediately consecutive one.
EAP	Earliest-arrival path	Path that can be completed in a given interval such that the ending time of the path is minimum.
LDP	Latest-departure path	Path that can be completed in a given interval such that the starting time of the path is maximum.
SP	Shortest path	The path that is shortest from x to y in terms of overall traversal time along the edges.
FP	Fastest path	A path that can be completed in a given interval such that its duration is minimum.

Table 9 Continuous paths experiments: Characteristics of each social network data set.