INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA

ESCUELA DE INGENIERÍA Y GESTIÓN

Temporal information query language for graph databases

AUTOR/ES: Debrouvier, Néstor Ariel (Leg. N° 55382)
Parodi Almaraz, Eliseo (Leg. N° 56399)
Perazzo, Matías (Leg. N° 55024)

DOCENTE/S TITULAR/ES O TUTOR/ES: Vaisman, Alejandro Ariel

TRABAJO FINAL PRESENTADO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN INFORMÁTICA

Lugar: Buenos Aires, Argentina
Fecha: 31 de julio de 2020

Temporal information query language for graph databases

Ariel Debrouvier, Eliseo Parodi Almaraz, and Matías Perazzo

Instituto Tecnológico de Buenos Aires, Buenos Aires, Argentina

Abstract. Many real word scenarios can be represented with graph databases such as friendship relations or brand following on a social network. In most of the cases, this information is not static and it varies over time.
The work on this paper applies temporal database concepts to graph databases, allowing to store, query and analyze time varying information. For this purpose, a suitable temporal graph data model was selected and a temporal query language, T-GQL, was developed to perform time varying data analysis. This language is similar to SQL and GQL, an upcoming standard graph query language at the time of writing this paper. Also it operates in cooperation with a collection of temporal algorithms, that where implemented, as well as T-GQL, over the widely known graph database Neo4j.

Keywords: Temporal Graph Databases · Neo4j · Query Languages · ANTLR · Cypher Query Language.

Table of Contents

1 Introduction and Motivation

Property Graphs [3, 17, 26] are becoming increasingly popular, particularly for modeling different kinds of networks for data analysis. The property graph data model underlies most graph databases in the marketplace [1]. Examples of graph databases and graph processing frameworks based on this model are Neo4j [1], Janusgraph[2], and GraphFrames [10]. Typically, the work of researchers and practitioners is based on graphs were the temporal dimension is not considered, called *static* graphs from here on. However, in real-world problems, time is present in most 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, property values can be updated, to mention the most relevant ones. For instance:

- (a) In a phone call networks, where each vertex represents a person (or a phone number), and an edge (u, v, t, λ) tells that u calls or sends a message to v at time t, and the connection time is 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 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 in a graph represents a location, and an edge (u, v, t, λ) represents a road segment, a street, or a highway segment, existing from u to v, since time t, 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 know 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), 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 "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 [6, 14], asking for the fastest way to reach a city in case a new highway is built.

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

Literature in temporal graphs is relatively limited, and basically oriented to address path problems particularly for problems of the kind of scenarios (a) and (d) above (see Section 2 for a discussion). As far as the authors are aware of, problems along the lines of the temporal databases theory [28] have not been addressed yet. 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., [30, 32]), where only edges are timestamped with the initial validity time of the relationship, and the duration of the relationship, and there is only one kind of relationship in the graph. In the model presented here, nodes and relationships contain attributes (properties), which are timestamped with a validity interval, and graphs are heterogeneous, that means, relationships may be of different kinds. These graphs are denoted Interval-labeled Property Graphs (ILPG) 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"*. 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 [25], and Consecutive Path Semantics. Both semantics, and how they are implemented, are discussed in detail.

More concretely, the contributions of this work are:

- A temporal graph data model for property graphs, allowing keeping the history of nodes, edges, and properties.
- A powerful graph query language, denoted T-GQL, based on GQL[15] (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, together with a client interface for querying Neo4j graphs.
- A collection of experiments over the implementation, over two cases covering the two semantics studied in this work: a synthetic dataset of a social network, and a real dataset of flights between several cities.

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 discusses the requirements of a query language for temporal property graphs. Section 5 introduces T-GQL, the high-level data language proposed for the model, and Section 6 present the implementation details. Section 7 reports preliminary experimental results, and Section 8 concludes the paper.

2 Related Work

This section reviews related work, 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 [1, 4]. The interested reader is referred to these works for details. Multiple native graph indexing methods and query languages (e.g., GraphQL [19]) 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, the structure of RDF makes them not as efficient as models in (b), which are aimed at reaching a local scope. On the other hand, RDF aims 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. In the *property graph* data model [3, 2], nodes and edges are labeled with a sequence of (attribute, value)-pairs. Extending traditional graph models, Property Graphs are the usual choice in modern graph databases used in real-world practice. Hartig [17, 18] 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, the standard query language for the Semantic Web. This is also studied in [5, 29].

2.2 Data models for temporal graphs

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

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

– (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. This paper will show that graphs of type (b) are more appropriate 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 by an initial database and a sequence of incremental updates.

2.2.1 Duration-labeled temporal graphs These kinds of graphs are studied by Wu et al. [30]. Graphs in that work are not property graphs, but simple graphs where a node is represented as a string, and the edges are labeled with a value representing a duration of the relationship between two nodes. Based on this work, the same authors have elaborated different proposals [20, 31–34]. All of them address the previously mentioned kinds of graphs. Definition 1 formally explains the above description.

Definition 1 (Duration-labeled graphs (cf. [30])). *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 $v \in G_d$, defined as $d_{out}(v) = \sum_{u \in N_{out}(v)} d(v, u)$ 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 [11]. In [30] (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

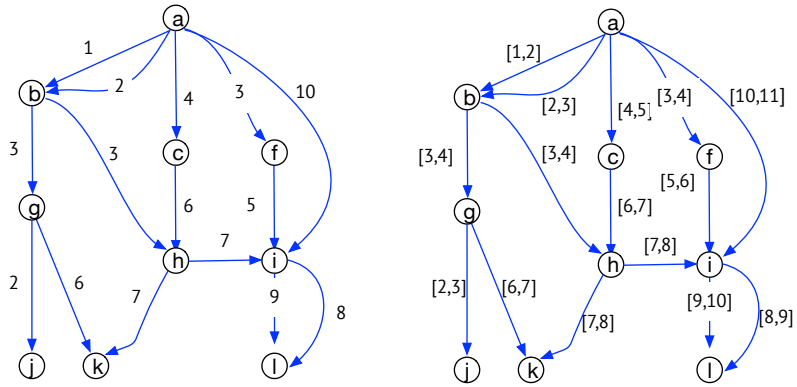

Fig. 1. Left: A duration-labeled temporal graph (cf. [30]); Right: An Interval-labeled graph for the graph on the left.

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 needed on the edges.

2.2.2 Interval-labeled temporal graphs There are two main approaches in the temporal databases literature [28], for keeping the history of a database: tuple or attribute-timestamping, where either 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. Note, however, that *these are still not property graphs.* 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 [7] (see Section 3), the first approach to apply the ILTG notion to property graphs. Valid time is considered in the remainder, that is, the times where the edges are valid in the real world, opposite to transaction time, which reflects the time where the information is stored in the database.

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.

Note that the interval-labeled representation supports different kinds of information, and are equivalent to duration-labeled temporal graphs (the proof is outside the scope of this paper). The difference is in the way in which temporal information is encoded. For example, to represent instant messaging each label may be timestamped with the interval $[t, t]$, meaning that the duration of the message is negligible. Social networks, travel schedules, trajectory representation can also be modeled in this way.

Example 1. The path traversal times defined 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 [30] 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 in equal to the departure time from b to g. □

Interval-labeled temporal graphs appear to be, at first sight, more appropriate than duration labeled graphs to support classic temporal queries, for example, the ones asking for the history of relationships in a social network. Moreover, current graph databases are based on the property graph data model, which are not supported in the work by Wu et al. Therefore, the data model in the present paper is based on interval-labeled property graphs, as well as the accompanying query language.

2.2.3 Temporal graphs as a sequence of snapshots The work by Semertzidis and Pitoura [27] 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 address 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, either contiguously (i.e., in consecutive graph snapshots) or collectively (i.e., in the largest number of graph snapshots). These queries are called *graph pattern queries*. Locating durable matches in the evolution of large graphs has many applications, like for example, long-term collaborations between researchers, durable relationships in social networks, and so on. In [27], a temporal graph is defined as follows, which defines the third category of temporal graphs defined above.

Definition 3 (Sequence-snapshot temporal graph). *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.*

Using graphs in this category, Huo and Tsotras [21] study the problem of efficiently computing shortest-paths on evolving social graphs. In this work, shortest-path queries are "temporal" in the sense that they can refer to any time-point or time-interval in the graph's evolution, and corresponding valid answers are returned. The authors extend the traditional Dijkstra's algorithm [11] to compute shortest-path distance(s) for a time-point or a time-interval. Different from traditional studies of shortest-path queries on a single graph, their main goal is to efficiently answer temporal shortest-path queries within the social graph's evolving history. Such temporal queries can be viewed as being 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. The authors define a temporal graph as an initial snapshot, followed by updates. Finally, several different kinds of path queries are defined. A *time point shortest path query* $TPSP(TEG, tq, vs, vt)$ 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,such that all edges in p are valid at query time t_q. Analogously a *time interval shortest path 'all'query* returns a set of paths P such that each path $p_i \in P$ is associated with a time interval Δ_{p_i}, and there is no other path shorter than p_i from v_s to v_t during Δ_{p_i}.

2.2.4 Other work on temporal graphs In the temporal graph model presented in [8, 9], 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 to retrieve the status of the social network during such interval. This model does not allow registering changes in attributes of the nodes. Also, frame nodes may become associated to 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 [22, 23] have studied 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. [16] presented an engine for temporal graph mining, and Kostakos [24] show the use of temporal graphs to represent dynamic events.

3 Data Model for Interval-labeled Property Graphs

The problem of extending property graphs in order to keep the history of the components of the model is addressed next. Property graphs are graphs such

that their nodes and edges are labeled with a sequence 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 [7]. The definition below is based on such 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 is 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) in which the node is (was) valid and it is a temporal element.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 `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 represent that the node is currently valid. Object nodes also have an identifier, denoted `id`.* □

In the definition above, 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 data 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, 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. □

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.

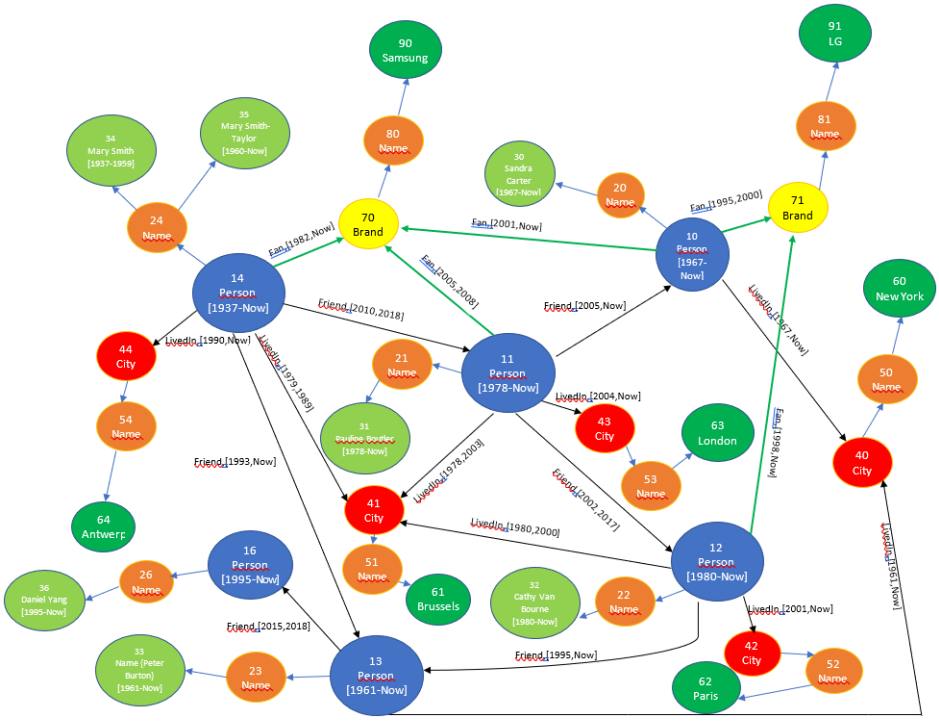

Fig. 2. A temporal graph and its different kinds of nodes

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 m edges 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. $\forall n_e\{n, n'\} \in N_e, n_e.interval \subset n.interval \cap n'.interval$
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 two nodes cannot have the same id. Constraint 4 requires coalescing all nodes with the same value; thus, the interval becomes a temporal element which includes all periods where the node had such value. The same occurs for edges and Constraint 5: 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 is stated by Constraints 9 through 10, stating 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. Constraints 11 to 14 restrict the values of the `interval` *property.* □

3.1 Continuous Paths

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 notions of *continuous path and maximum continuous path* [25]. Definition 7 introduces these concepts.

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 $(n_1, \ldots, n_k, T)$ of k nodes and an interval T such that there is a sequence of consecutive edges of the form $e_1(n_1, n_2, r, T_1)$, $e_2(n_2, n_3, r, T_2)$, ..., $e_k(n_{k-1}, n_k, r, T_k)$, and $T = \bigcap_{i=1,k} T_i$.* □

Example 3 (Continuous Path). Consider the graph depicted 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, namely $(n_1, n_2, n_3, n_4, friend, [2, 3])$ 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 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 Paths

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

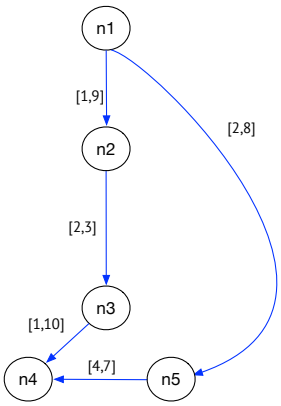

Fig. 3. Continuous paths and maximal continuous paths

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, note that pairwise, such intersection exists, e.g., there is an overlap between $(p_1, p_2, friend, [2, 7])$ and $(p_2, p_3, friend, [6, 12])$. That means, although there is not a continuous path between p_1 and p_4, there is a consecutive chain of pairwise temporal relationships in the path. This is formalized by the notion of *pairwise continuous path*.

Definition 8 (Pairwise Continuous Paths). *Given a temporal property graph G, there is a* pairwise continuous path *between two nodes n_1, n_k, through a relationship r, if there 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 of 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 nodes $P = (n_1, n_2, r, [t_1, t_2]) \ldots, (n_{k-1}, n_k, r, [t_{k-1}, t_k]))$ 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 $1 \leq i \leq k$. The 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 [30]. The ones that will be studied in this paper are introduced by 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.

4 Temporal Query Language Requirements

The model and path concepts defined in Section 3 set the basis for defining a high-level query language for temporal graphs. This section introduce the classes of queries that such language must address. The detailed syntax and semantics are given in Section 5.

4.1 Interval-based Queries

The presentation starts with queries referring to the social network example in Figure 2, which refer to continuous path queries, and queries that take advantage of the interval-labeled model. The query language addressed different types of queries, which are informally classified as follows.

4.1.1 Continuous path queries These queries compute paths valid continuously in an interval, following the semantics of Definition 7. In addition, temporal filters can also be applied over the interval of the continuous path. The FOF query below is a typical case of this kind of query.

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

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 answer to Query 1 will include the path $Mary \rightarrow Pauline \rightarrow Cathy, [2010, 2017]$ since the whole path was valid in this interval (this is the continuous path of Definition 7).

As another example, where a filter over the continuous path interval is included, is shown next.

Query 2 *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; one of length 3 and the other of length 1, discarded by the lower bound in the filtering condition. 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 $[2010, 2017] \cap [2018, 2020] = \emptyset$.

4.1.2 Queries in parallel periods These kind of queries make use of the power provided by the labeling model.

Query 3 *Who were friends of Mary while she was living in Antwerp?*

Mary lived in Antwerp between [1990-Now]. The adopted semantics for this query is that the answer would be any person that was a friend of Mary *at any instant* of that interval.

Query 4 *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 must allow 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. It can be noticed that the former two queries would be much difficult and unnatural to express with a duration-labeled representation.

4.1.3 Snapshot queries

Snapshot queries use the notion which is usual in temporal databases, that is, a snapshot is a non-temporal database valid at a certain time instant. Thus, these queries should return a non-temporal graph.

Query 5 *Who were the friends of the friends of Cathy in 2018?*

This query would return only those people which during any moment of the year 2018 maintained a friendship with a friend of Cathy. Additionally, the graph will also show the direct friends of Cathy during that year.

4.1.4 Queries asking for interval conditions

Query 6 *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.

4.1.5 Pairwise consecutive path queries

The notion of consecutive path must also be supported by the language. For the social network running example, the typical "degree of separation" query is an example of this. Since, of course, in this example, the people in the path would have a link stronger that the popular handshaking condition), this is called "friendship degrees of separation", meaning that the friendship relationship is required to be valid in a pairwise fashion and not throughout the whole interval.

Query 7 *List the people with less than six friendship degrees of separation from Mary.*

4.2 Consecutive Path Queries

In addition of the above queries typical of ILTGs, the paper aims at showing that the queries that the duration-labeled temporal graphs support, are also supported by ILTGs, through the notion of consecutive path, and a collection of temporal functions, as it will be shown in the next sections. To show this, another example will be used. In this example, there two object nodes are Airport and City. The temporal relationships are 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. Note here the flexibility that the ILTG model provides. Also, it is worth noting that this does not intend to be a real-world example of flight scheduling, of course. A portion of this example is depicted in Figure 4.

Queries over this example are studied next. In these kinds of queries, the notion of continuous path cannot be applied, as it will be shown.

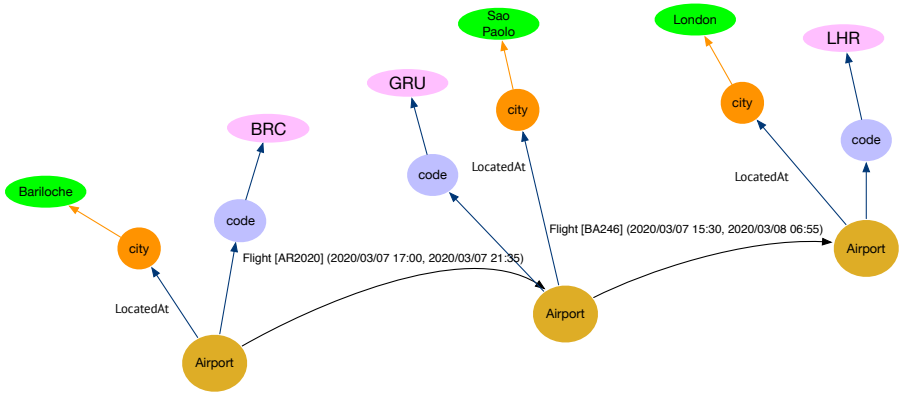

Fig. 4. A temporal graph for flight scheduling analysis.

Query 8 *How can we go from Tokyo to Buenos Aires as soon as possible?*

Recalling Definition 10, this query refers to the earliest-arrival path (EAP) 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, and allows showing the flexibility of the interval-based model. A path in the solution must be such that the intervals of the edges are pairwise disjoint.

Examples of other queries addressing Definition 10 follow.

Query 9 *How can we go from Tokyo to Buenos Aires, leaving as late as possible and arriving before July 15 at 8 pm?*

For this query, the latest-departure path semantics solves the problem.

Query 10 *How can we go from Tokyo to Buenos Aires flying the least possible number of segments?*

This query refers to the shortest path.

Query 11 *How can we go from from Tokyo to Buenos Aires as fast as possible?*

The fastest path interpretation for this query, would be the one in which the whole trip, including layovers, would require less time.

5 T-GQL Syntax and Semantics

The requirements studied in the previous section give the rationale of the high-level query language that will be described next. The language syntax is based on GQL [15], a standard being defined at the time of writing this paper. The language has an SQL flavor, although it is based on Cypher[4], Neo4j's high-level

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

query language. The formal semantics of Cypher can be found in [12, 13]. Cypher was extended with a collection of functions, whose implementation is explained later in Section 6 below.

5.1 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 one or more path patterns (of fixed or variable length) and function calls. The result of the query is always a temporal graph (analogous to relational temporal databases theory), although the query may not mention temporal attributes. This can be modified by the SNAPSHOT operator, which allows retrieving the state of the graph at a certain point in time. The syntax and semantics will be introduced using the social network in Figure 2. Path procedures (Definition 10 will be covered using the flight example.

The first query simply asks for object nodes: "List the friends of the friends of Mary Smith-Taylor", expressed in T-GQL as:

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

This query above does not include temporal features (the MATCH clause does not ask for temporal paths), but allows introducing the basic T-GQL syntax. Temporal capabilities are addressed in the next sections. The query 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, maybe using an alias, as follows:

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

In order to select all the paths involved in the MATCH clause, the wildcard operator '*' is allowed. 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'
```

5.2 Temporal Path Functions

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.

Remark 1. Functions computing coexisting (continuous and pairwise) paths, do not accept the term '*'. 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, since these functions return a unique path, therefore '*' must be used.

5.2.1 Continuous paths and interval-based queries Query 1 is an example of the use of coexisting path function. Recall, this query asks for the friends of the friends of each person, and the period such that the relationship occurred continuously through all the path (i.e., the continuous paths). 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:

```
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. For example, to find the continuous paths between Mary Smith Taylor and Peter Burton with a minimum length of two and a maximum of three, a user could write:

```
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.

paths


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


It can be seen that here, the attribute and value nodes are embedded in the answer, to facilitate their search (as mentioned, object nodes are not likely to be useful for a final user). Note that the value node "Mary Smith" is ignored since it's 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 $[1960-Now]$ and $[2010-2017]$. Finally, the interval of the continuous path is $[2010-2017]$, which is the result of the intersection between the traversed edges $([2010-2018], [2002-2017], [1995-Now])$.

The cpath function is overloaded to also return a Boolean value. For example, to get the names of the persons such that there is a continuous path from them to Peter Burton, the T-GQL query would read:

```
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 guesses from the context that the Boolean procedure must be used.

Pairwise continuous paths can be also computed, using the pairCPath function. Since they ask for pairwise intersection intervals, the function computing these paths is denoted pairCPath. An example is shown below.

```
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, given by the user. This will filter out the paths whose interval does not intersect with I. The granularity of the starting and the ending instants of the interval must be the same. The query below (Query 2) filters the paths with the interval [2018-2020] (in this case the answer is empty, because the intersection of the intervals [2010-2017] and [2018-2020] is empty):

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

5.2.2 Consecutive path queries For explaining consecutive paths, the example in Figure 4 will be used. Four functions are supported: fastestPath, earliestPath, shortestPath, and latestDeparturePath. The first three functions receive

two nodes as arguments (recall, '*' is mandatory for these path functions, unlike the case of continuous paths). The latter also receives a time instant. Examples are shown next. As an example, Query 11: *How can we go from Tokyo to Buenos Aires as fast as possible?*, reads in T-GQL:

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

As mentioned, the latestDeparturePath function needs a threshold parameter as argument. As an example, Query 9: *How can we go from Tokyo to Buenos Aires, leaving as late as possible and arriving before July 15 at 8 pm?*, is written as follows.

```
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 = 'Tokyo' AND c2.Name='Buenos Aires'
```

5.3 Handling Temporal Granularity

The reader may have noticed that all time intervals in the social network example are given in 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 coarser one is transformed into the finer 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 $o_g.interval = [1/1/2010, 31/12/2012]$.
- 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.

As mentioned, in the social network example, the granularity used in year, for all data. Also the queries where expressed using this granularity, so no problem arouses 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 and formats:

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

Examples will be presented in the next sections.

5.4 Temporal Operators

Temporal operators and filters are explained next. These are used to answer some of the types of queries introduced in Section 4.

On the one hand, there are the SNAPSHOT operator, which receives as argument a temporal value, and the BETWEEN operator, which receives a temporal interval. Both of them perform an explicit temporal filter since a temporal argument is given explicitly. On the other hand there is the WHEN clause, which is similar to the concept of an SQL inner query. It has the form MATCH-WHERE-WHEN and can have references to variables from the outer query. Function calls are not allowed within this clause, and it can only handle exactly one two-node path and a single edge in its MATCH clause.

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. For example, Query 5: *Who were the friends of the friends of Cathy in 2018?*, is expressed as:

```
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 int 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 that 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 the start and end must be the same. For example, Query 6: *Where did the friends of Pauline live between 2000 and 2004?* reads:

```
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 filter helps to answer parallel periods queries. Query 3: *Who were friends of Mary when she was living in Antwerp?*, could be expressed as:

```
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). Basically, the inner query is executed, returning a collection of intervals. Then, the WHEN performs a BETWEEN for each of these intervals.

6 Implementation

This section describes the implementation details 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.

6.1 Architecture

The model and language described in this paper were implemented over the Neo4j temporal database, an open-source Java-based graph database. Neo4j allows extending its functionality by creating 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.

As mentioned in previous sections, the T-GQL language is based on the standard being developed, GQL. The language grammar was implemented using the widely-known tool ANTLR[5]. Using 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 5 sketches the system's architecture. To edit and execute T-GQL queries, a web application interface was developed, also coded in Java, using the lightweight web framework Javalin[6]. The application exposes a page where

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

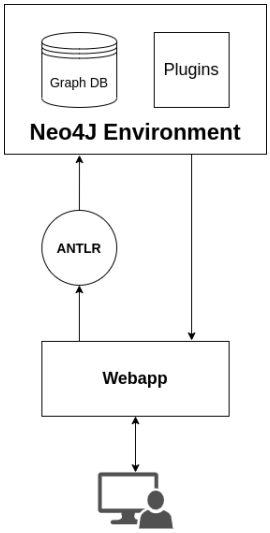

Fig. 5. General architecture.

the queries can be executed from an endpoint. This tool uses the main parser to translate the users' queries into Cypher and execute it on a Neo4j server, that contains the plugins to run the temporal operators and path algorithms.

In addition, for populating the two running example databases (and also for the experiments reported in the next section), a dataset generator was developed. Parameters for this generator allow indicating the model to be populated, as well as the number of relationships and nodes, and number of intervals that each edge can have, among many 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.

6.2 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 6. 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

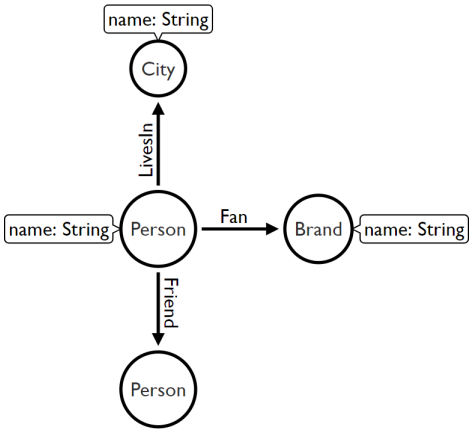

Fig. 6. Social network metamodel

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 7. 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, but also 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 8 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 with the 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 as output. 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

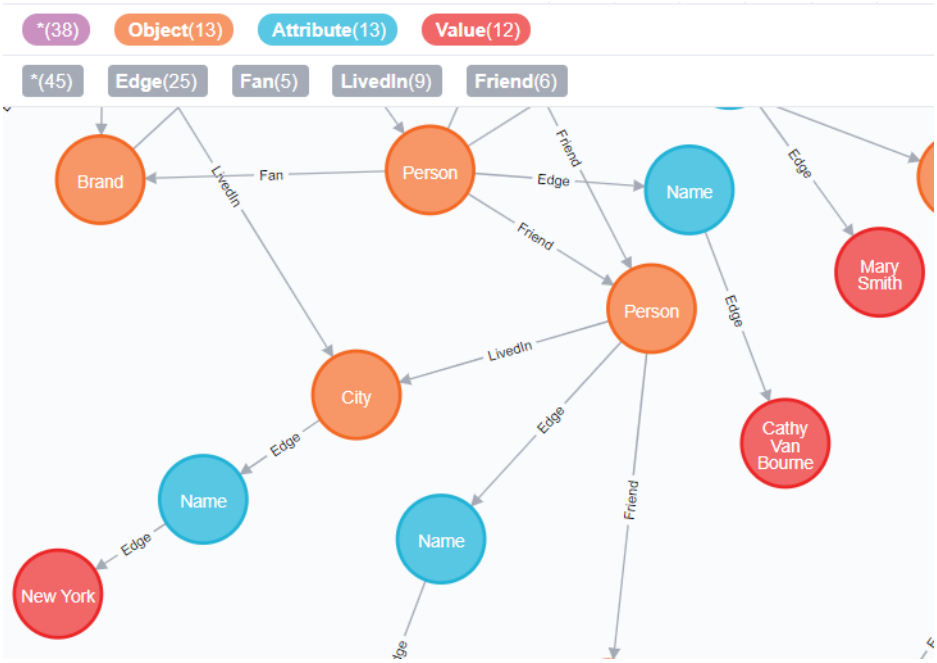

Fig. 7. Social network model for the metamodel in Figure 6

{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 is shown at the end of this section.

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
```

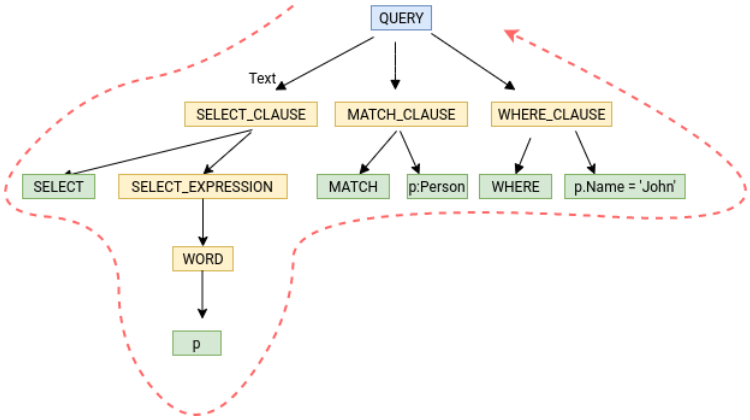

Fig. 8. Example parse tree

Queries mentioning functions are explained next. Consider the following query, asking for a continuous path:

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

The temporal procedures (like `coexisting.coTemporalPaths`, used in this query) are described in Section 6.3 below. 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 cities have 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. To retrieve the fastest path, the `paths.intervals.fastest` aggregation function is called. It receives all the paths and returns only the fastest ones, according to Definition 10.

6.3 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 considerable 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. In this work, Between and Snapshot are defined.
- *Coexisting paths procedures:* Implement the continuous path semantics (including the pairwise continuous paths).
- *Consecutive paths procedures:* Implement the consecutive path semantics.

The procedures above are packed in a library analogous to the neo4j-graph-algorithms library.[7] The *Coexisting* and *Consecutive* procedures extend a framework defined to work on temporal graphs. This framework was based on the neo4j-graph-algorithms library[8]. This library contains some graph implementations of graph algorithms. However, there are no algorithms for temporal graphs.

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.

Coexisting paths procedures These procedures return the continuous paths of a given length, either starting from a node, or between two nodes. The procedures also have a Boolean alternative, that can be used for checking whether or not a path exists between two nodes, for example. 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 $f(i1, i2) = i1 \cap i2$. That way the intersection of the intervals are only stored, and the algorithm keeps iterating with the intersections. For pairwise temporal paths, f is defined as $f(i1, i2) = i2$, this way it only returns the latter interval, and the algorithms 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. Until the queue is empty, the algorithm picks up the paths in the queue. 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, the algorithm 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

[7] https://github.com/neo4j-contrib/neo4j-graph-algorithms
[8] 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(new_tuple)
      if L_min <= length + 1 <= L_max and (y not exists or dest == y) then
        S.insert(copy)
      end if
      if length + 1 < L_max then
        Q.enqueue(copy)
      end if
    end if
  end for
end while
```

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 solutions S. Otherwise, the path 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*.

Consecutive paths procedures These procedures are based on the graph transformation approach introduced by Wu et al. [32] for DLTGs, which was adapted to address the temporal graph model of Definition 4. However, unlike the approach presented in [32], the algorithm presented here prevents creating the whole graph to apply the path computation algorithms, since this would be extremely expensive. Instead, in the present proposal the transformed graph is built as the iterations proceeds over the original temporal graph, call it G. The transformation creates a new graph, denoted G_t, where the nodes contain either the starting

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 L_min <= length + 1 <= L_max and (y not exists or dest == y) then
        return true
      end if
      if length + 1 < L_max then
        copy = current.copy()
        copy.insert(new_tuple)
        Q.enqueue(copy)
      end if
    end if
  end for
end while
return false
```

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 and edge is the duration of the corresponding interval. This new graph is easier to iterate as it 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. For clarity, these attributes are denoted (for a node n), $n.noderef$, $n.time$, $n.length$ and $n.previous$ in Algorithm 3.

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
```

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 v_t (the start time of the interval in the edge), the length of e plus 1, and e as the previous node, that means $(v_t, t.start, e.length+1, null)$. The node v_{in} contains 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 functions 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_t, 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_t 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

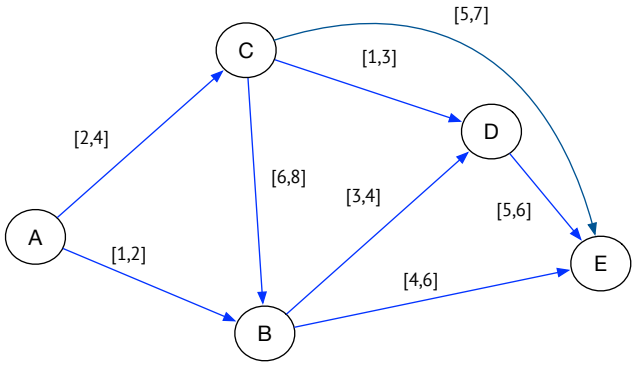

Fig. 9. An example for Algorithm 3

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 that was 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 9 shows a graph over which the shortest path between nodes A and B will be computed with Algorithm 3. The function f will thus be $f(x, y) = x.length - y.length$.

The first node created in G_t is $(A, -\infty, 0)$ (the reference to the previous node is omitted, for clarity), which will be 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 9 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, $[1, 3]$ 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 lead to the shortest paths A, B, E and A, C, E.

7 Evaluation

A collection of experiments were developed in order to test the different algorithms described and implemented in this work.

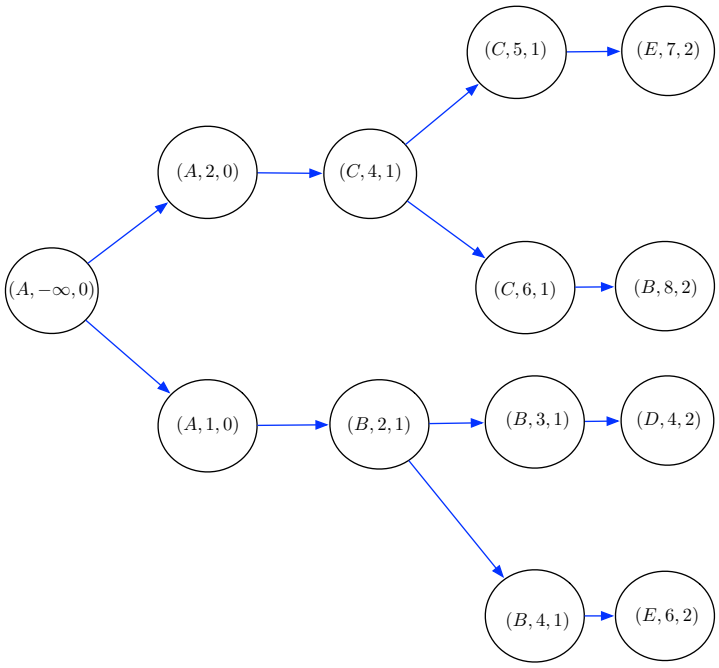

Fig. 10. An example for Algorithm 3

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

7.1 Use cases

7.1.1 Continuous paths algorithms The objective of the experiments were to test how did the length of the paths and the size of the dataset impacted on the performance of the algorithm. Therefore, different tests were conducted, varying both variables.

The queries involved perform a continuous path search of a given length between two specific persons, using an id property generated during the population of the dataset. For example, the following query obtains all the Continuous Paths of length 8, between the Person nodes with id 10 and 30, and would serve as a query template for the experiments.

```
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 developed for the Pairwise Continuous Algorithm:

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

7.1.2 Consecutive paths algorithms The objective of the experiments were to evaluate how do the different paths behave with distinct graph sizes. Also, as it was worked with a dataset of real-life flights, some airports with many connections and some with few were chosen to appraise how the algorithm works with a combination of these.

The queries involved perform a consecutive paths search for two specific airports using their IATA (Internation Air Transportation Asociation) code, that is a code that consists of three letters that identifies an airport.

```
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'
```

7.2 Datasets

7.2.1 Continuous paths algorithms For testing Continuous Path and Pairwise Continuous Path algorithms, a dataset generator based on the model described on Definition 4 and represented in Figure 2 was implemented. This generator allowed to populate a Neo4j database specifying the following parameters:

- N = Number of Person nodes
- F = Maximum number of Friendship relationships per person
- I = Maximum number of intervals per friendship
- Number (C) and length (L) of Continuous Paths

First the generator creates C continuous paths of length L and then generates the friendships for the whole graph. Furthermore they are generated randomly, so it can be said that the generator ensures a minimum of C continuous paths of length L.

As soon as the generation ends the id of the persons implicated in each continuous path are logged and stored. Therefore a query can be performed with the template above using the ids of the start and end of the path, and the L as the variable length limiter of the function call. Consider that more than a path could be obtained as a result, as a consequence of the random generation of friendships mentioned above.

The execution of a query for a specific N and L is carried C times varying the ids of the start and end nodes of the path. This approach is taken because the time required to run the query can vary depending on the quantity of continuous paths of length L found and the nodes visited. So C measurements are performed obtaining the execution time and the paths found for each pair of nodes. In the following section it will be discussed how these indicators were used to evaluate the results.

Three datasets were generated, with N = 1000, 10000 and 100000 and fixing the remaining parameters to F = 5 and I = 2. For each dataset, 3 paths (C) of each of the following lengths (L) were generated: 4, 6, 8, 10 and 12.

In Table 1 it is specified the number of nodes, edges and byte size of each of the datasets. Indexes were created on the Object, Value and Attribute nodes for the id property, since it enhanced the performance of the dataset creation, but not any other index was created.

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

Table 1. Dataset details for each social network dataset.

7.2.2 Consecutive paths algorithms Consecutive path algorithms were tested using a real world flight database, the Flight Delays and Cancellations for US flights in 2015.[9], using the original departure and arrival times for the flights.

Five datasets were generated from this dataset filtering the flights by different time intervals. The selected periods to create the datasets were the first week, first month, first three months, first half and the entire year. For each of this datasets, the number of flights and airports are shown in Table 2.

This may look like a little number of airports but the important part is the number of nodes of the transformation graph that are generated. For each

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

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

Table 2. Number of airports and flights in each dataset.

dataset, is taken into account the number of V_{out} and V_{in} nodes of the complete transform graph. The values of the number of nodes are in Table 3.

Dataset	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 dataset.

From this datasets some aiports in particular were chosen.

1. ATL - Hartsfield–Jackson Atlanta International Airport, Atlanta, Georgia.
2. CLD - Mc Clellan-Palomar Airport, Carlsbad, California.
3. BOS - General Edward Lawrence Logan International Airport, Boston, Massachusetts.
4. HOU - William P. Hobby Airport, Houston, Texas
5. SBN - South Bend Regional Airport, South Bend, Indiana
6. ISP - Long Island Mac Arthur Airport, Islip New York

Different paths were selected between the previously mentioned airports, mixing different sizes of airports.

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

A path between two large airports was not selected because usually there are direct flights between them, this means a path of length 1 exists and therefore the performance and potential of the implemented algorithms would not be tested thoroughly.

The number of incoming and outgoing flights are listed in Table 4. It is noticeable that the airport CLD number of flights stops growing at the 6 months

as the airport closes. This airport was chosen as it challenges the behaviour algorithm of latest departure path, as it will try to search for the latest departure path going to the paths with the latest departure time but the arrivals are all in the first half of the year.

Airport	Dataset	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 4. Total number of nodes in each dataset.

7.3 Results

Due to the fact that there are different indicators involved in the complexity of the search of a continuous path, it is difficult to compare only the execution time of the algorithm between different queries. Therefore, instead of averaging the execution time, the following average definition was used, were n is the total number of pairs of nodes (start and end of a continuous path) involved in the query, t the execution time and c the number of paths found for each pair of nodes.

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

For example, if a dataset is generated with $C = 3$, three continuous paths of length L are generated. One between node A_1 and node A_2, and identically between the pairs of nodes A_3 and A_4 , and the pairs A_5 and A_6.

In Table 5, a set of example results of query execution are shown.

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

Table 5. Example results for continuous path query execution.

Therefore, the average T can be calculated with the following formula:

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

This average definition is used for Figure 11 and Figure 13. On the other hand, for Figure 12 other average is applied by dividing the execution time by the number of paths found for each pair of nodes.

Finally for the consecutive paths, the normal definition of average is used, running the algorithms three times for each path and dataset. The execution times are summarized and divided by three. The number of results are also taken in consideration. This are the sum of the number of intervals from the algorithm for each path retrieved, as they are the number of paths generated in the transformation graph.

7.4 Discussion of Results

7.4.1 Continuous Paths Figure 11 represents the execution time for each dataset size, and different continuous path lengths. For $N = 10000$ and 100000, the execution time increases along with the length of the path searched: it starts with values around 50 ms for $L = 4$ and growths into 733 ms and 3279 ms

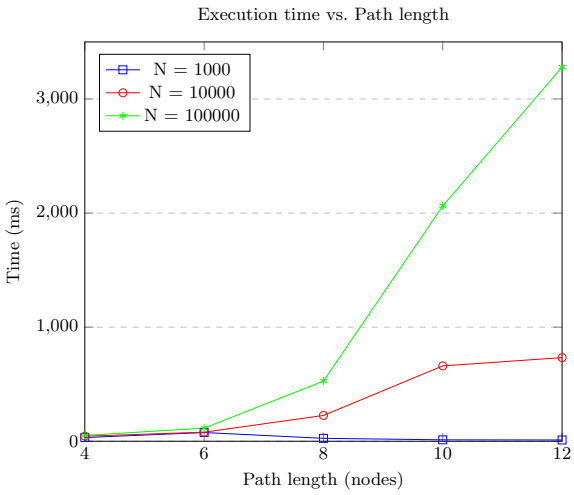

Fig. 11. Execution time vs. Path length for continuous path algorithm

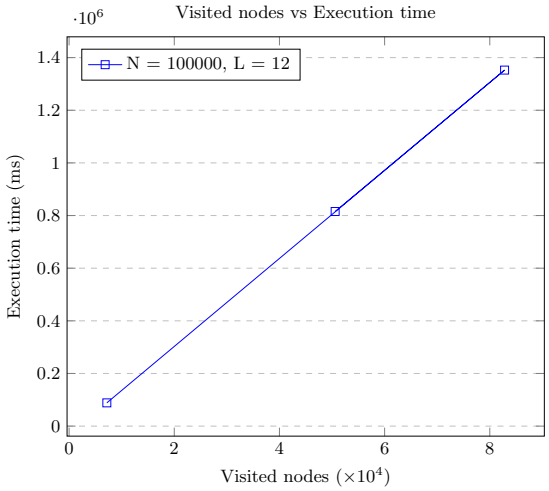

Fig. 12. Visited nodes vs. Execution time for continuous path algorithm on paths of $L = 12$. The visited nodes are the nodes that are traversed for a continuous path search. Each point is a division between the execution time and the paths found for a pair of nodes.

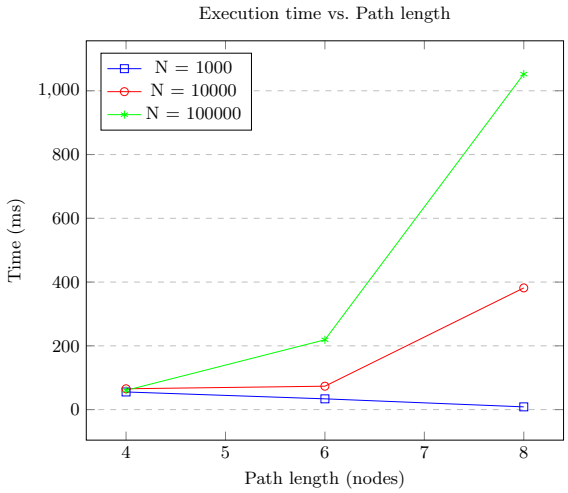

Fig. 13. Execution time vs. Path length for pairwise continuous path algorithm

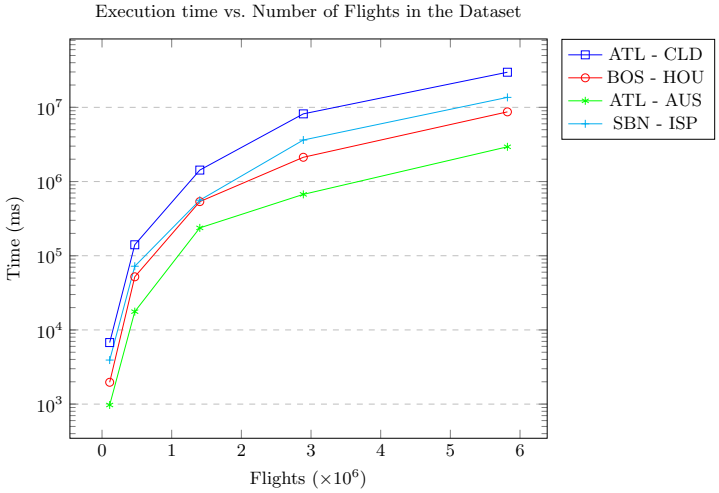

Fig. 14. Execution time for each pair of airports for the fastest path algorithm.

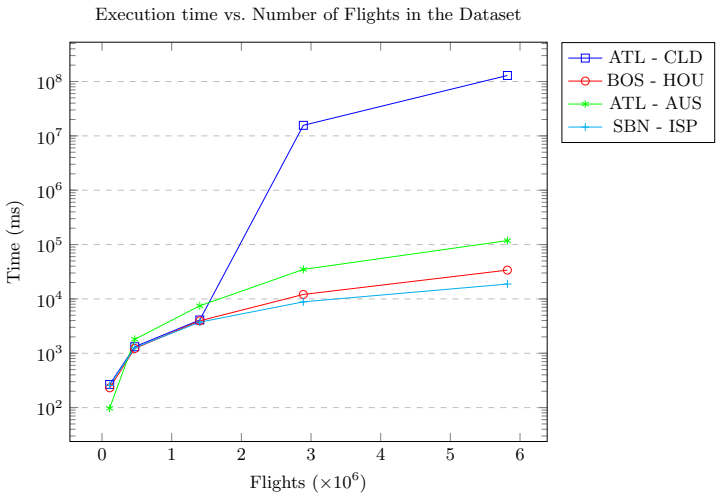

Fig. 15. Execution time for each pair of airports for the latest departure path algorithm.

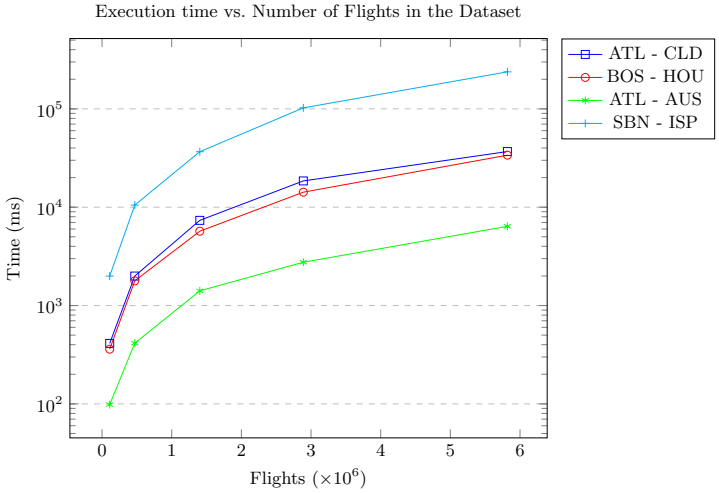

Fig. 16. Execution time for each pair of airports for the earliest path algorithm.

Path	Dataset	Earliest Path	
		Average Time (ms)	N. of 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.

respectively for L = 12. On the other hand, for $N = 1000$, the growth of the execution time is steadier, it starts with an execution time of 30 ms and even decreases for longer paths, without exceeding 80 ms for any case. For the dataset with $N = 100000$ the growth of the data is similar to an exponential function.

By analyzing the number of visited nodes by the algorithm in Figure 12 and comparing it to the execution time, the results are similar to a linear function, proving that visiting nodes increase the execution time. This happens due to the fact that traversing the graph requires memory and processing usage, increasing the execution time of the algorithm.

The results for the pairwise continuous paths were similar to the continuous paths ones: increasing the length of the path searched implies higher execution times.

7.4.2 Consecutive Paths The graphics in 14 for the fastest path, the 15 for the latest departure path, 16 for the earliest path and the 17 for the shortest path show a lineal behaviour in most of the cases of the algorithm in relation with the number of flights in the dataset that is represented in the x axis. The y axis is logarithmic as the difference between the running time of the algorithms is very different depending on the path. As expected, the execution time of the algorithm grows as the amount of flights grows.

Path	Dataset	Shortest Path	
		Average Time (ms)	N. of 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.

In the latest departure path, all the paths show a lineal behaviour but 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 find a path between those airports so it can prune the graph. That is why the time growths exponentially and then continue with a linear behaviour. The same behaviour is expected for the earliest path with a path where all the flights are in the last half of the year. This kind of obstacle cause the algorithm to run in 4098ms for the dataset of 3 months to 15622280.67ms for the one of 6 months. This is about 3800 times the time of the previous dataset. For this other airports, the time of execution in the larger dataset is approximately 3 times larger than the one in the smaller. This shows that this is a big disadvantage for this algorithm, but could be solved executing a similar BFS algorithm.

In the case of the shortest path, the number of results of the query made it impossible to finish for the biggest dataset. The out of memory was caused by the number of paths that are stored in the memory of certain length of below. In the case of the two smallest airports, there a lot of paths between this airports and with a relevant length. There is also a problem where a path can start in the beginning of the year and end at the end of the year. In the future, it would be a great improvement to limit this time for dataset that occur in a long time span.

Path	Dataset	Fastest Path	
		Average Time (ms)	N. of 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 8. Average time and number of results for the fastest path algorithm.

For the algorithms, with the exception of these cases of shortest and latest departure path, the execution times are similar, depending on the algorithm, some paths have better performance over the others but it depends on the algorithm and in the graph and not in the type of airports chosen. The algorithm with the lowest performance, besides the problem with the latest departure path from ATL to CLD, is the fastest path. For example, for the largest dataset, for the paths from BOS to HOU and ATL to AUS, the average execution time was 8694658.33ms and 2938933ms respectively. For the earliest path, this times were 33812ms and 6391.67ms, for the shortest path, 468ms and 1268ms, and for the latest departure 33875.33ms and 118174.67ms. The difference in the order of magnitude is large. But for the smallest dataset the difference of the execution times between the algorithms is not significant.

This may be caused by the nature of this paths, as it depends on the difference of the times of the first and last node of the path. The earliest departure path depends on the time of the last node of the path, the latest departure, 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, if a path reaches a node and has a latest departure path possible, 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, we could find a fastest path because the difference of time varies depending on the first and last node, and there is more variety because

Path	Dataset	Latest Departure Path	
		Average Time (ms)	N. of 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 9. Average time and number of results for the latest departure path algorithm.

the flight times are different between airports, not as in the case of the shortest path where the step between one node and the next in the path is of 1 in the length of the path. This can make the algorithm explore a node many times, increasing the execution time. That is the reason why the time of this algorithm varies significantly with the time of other algorithms.

8 Conclusions

The temporal property graph data model described in this work, along with the graph query language T-GQL proved to be a powerful tool for querying time varying data on graph databases.

For the algorithms developed for finding continuous and consecutive paths, the size of the graph along with the length of the paths being searched, are important factors to consider for the performance of the queries. In the case of a shortest path search for a big dataset with many relationships between the nodes, the algorithm can consume a lot of resources, leading to an out of memory error.

The flexibility of both the query language and the developed algorithms allowed to apply them on two different study cases, a social network and a flight schedule, and has to potential to be applied on many more, allowing to discover more use cases and purposes.

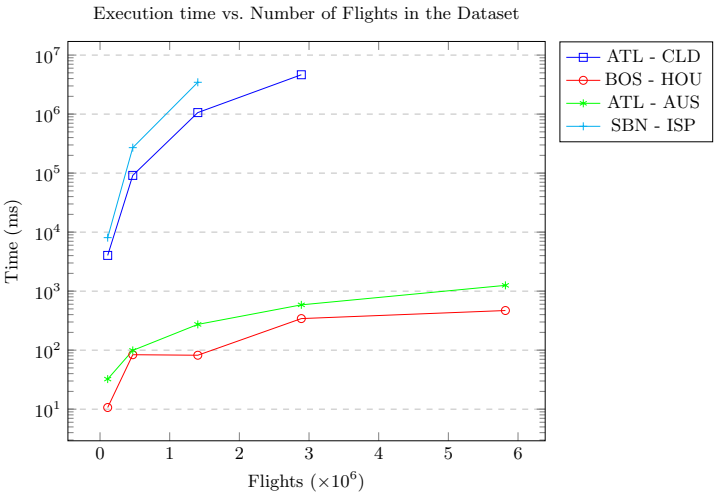

Fig. 17. Execution time for each pair of airports for the shortest path algorithm.

Future work would include research on indexing the time-varying data, therefore enhancing the performance of both the continuous and consecutive path algorithms.

The consecutive path algorithms could be improved by storing an interval or the length of the shortest path between two nodes, and then use it to prune the graph in the algorithm, reducing the consumption of resources and the execution time. Furthermore the search could be restricted by a certain time interval or length, or by the nodes contained in the path and their order.

The query language T-GQL, also has possibilities of improvement, expanding its grammar and adding useful functions or clauses that Neo4j provides, but for the scope of this work were not implemented. Moreover the $WHEN$ clause could be improved so it can have a path function call or more than a path in the $MATCH$ clause, adding more power to the language. Also it would be useful to have multiple $WHEN$ clauses in a query, which is not supported currently.

References

1. R. Angles. A Comparison of Current Graph Database Models. In *Proceedings of ICDE Workshops*, pages 171–177, Arlington, VA, USA, 2012.
2. 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.
3. 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.
4. Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
12. 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.
13. 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.
14. Matteo Golfarelli and Stefano Rizzi. What-if simulation modeling in business intelligence. *IJDWM*, 5(4):24–43, 2009.
15. Alastair Green. Gql - initiating an industry standard property graph query language, 2018.
16. 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.
17. O. Hartig. Reconciliation of RDF* and property graphs. *CoRR*, abs/1409.3288, 2014.
18. O. Hartig. Position statement: The rdf* and sparql* approach to annotate statements in rdf and to reconcile rdf and property graphs, 2019.
19. 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.
20. Silu Huang, James Cheng, and Huanhuan Wu. Temporal graph traversals: Definitions, algorithms, and applications. *CoRR*, abs/1401.1919, 2014.
21. 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.
22. U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. *CoRR*, arxiv:1207.5777, 2012.
23. U. Khurana and A. Deshpande. HiNGE: Enabling Temporal Analytics at Scale. In *Proceedings of SIGMOD*, NY, USA, 2013.
24. V. Kostakos. Temporal graphs. *CoRR*, arxiv:0807.2357, 2008.
25. F. Rizzolo and A. Vaisman. Temporal XML: Modeling, indexing, and query processing. *VLDB Journal*, 1179–1212(5):39–65, 2008.
26. I. Robinson, J. Webber, and Emil Eifrém. *Graph Databases*. O'Reilly Media, 2013.
27. Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Trans. Knowl. Data Eng.*, 31(1):181–194, 2019.
28. A. Tansel, J. Clifford, and S. Gadia (eds.). *Temporal Databases: Theory, Design and Implementation*. Benjamin/Cummings, 1993.
29. Harsh Thakkar, Renzo Angles, Dominik Tomaszuk, and Jens Lehmann. Direct mappings between RDF and property graph databases. *CoRR*, abs/1912.02127, 2019.
30. Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.
31. 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.
32. 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.
33. 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.

34. 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.